NTNU
Det skapende universitet

Path Rasterizer for OpenVG

Eivind Lyngsnes Liland

Master i datateknikk
Oppgaven levert: Juni 2007
Hovedveileder: Morten Hartmann, IDI
Biveileder(e): Lasse Natvig, ARM
Mario Blazevic, ARM
Thomas Austad, ARM

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

OpenVG (Open Vector Graphics) is a new standard API for drawing 2d vector graphics with support for hardware-acceleration and handheld devices. Graphic images are drawn using paths of connected segments. Types of segments include quadratic curves, cubic curves and arcs. The interior as well as the outline (stroke) of each path can be filled using a variety of techniques and patterns.

A common way to render interiors and strokes is to approximate them using polygons consisting of a large number of very short line segments. These polygons are then triangulated and finally rendered. The conversion to polygons and the triangulation is normally done on the CPU, since these operations are not easily implementable in a traditional GPU pipeline. The resulting triangles are rendered using the GPU.

A large number of long, thin triangles usually result from the conversion and triangulation processes. This is not rendered efficiently on most immediate mode renderers, and is even more problematic on most tile-based renderers. Also, CPU overhead from conversion and triangulation can be a significant bottleneck.

The goal of this project will be to find and implement an efficient and robust algorithm for rendering OpenVG paths that minimize or eliminate these performance problems. The algorithm may require minor additions or modifications to existing GPU hardware, or it may be a pure software solution.

The subtasks of the project are:
1. Describe an algorithm for rendering the interior of paths efficiently, with support for all possible fill
techniques, using an OpenGL ES 2.0 conformant GPU, with possible hardware additions and modifications.
2. If time permits, one or more of the following tasks can also be included:
  A) Implement and test the algorithm for rendering path interiors.
  B) Describe an algorithm for rendering strokes (path outlines), with support for all possible fill techniques, so that both interiors and strokes can be rendered efficiently.
  C) Implement and test the algorithm for rendering strokes.
  D) Make the algorithm work for an OpenGL ES 1.1 conformant GPU, with possible additions and modifications to the hardware as well as the algorithm.

Responsible at NTNU and main supervisor: Morten Hartmann
Co supervisors: Lasse Natvig at IDI, Thomas Austad and Mario Blazevic at Falanx/ARM.

Oppgaven gitt: 24. januar 2007

Abstract

Vector graphics provide smooth, resolution-independent images and are used for user interfaces, illustrations, fonts and more in a wide range of applications.

During the last years, handheld devices have become increasingly powerful and feature-rich. It is expected that an increasing number of devices will contain dedicated GPUs (graphics processing units) capable of high quality 3d graphics for games. It is of interest to use the same hardware for accelerating vector graphics.

OpenVG is a new API for vector graphics rendering on a wide range of devices from desktop to handheld. Implementations can use different algorithms and ways of accelerating the rendering process in hardware, transparent from the user application.

State of the art vector graphics solutions perform much processing in the CPU, transfer large amounts of vertex and polygon data from the CPU to GPU, and generally use the GPU in a suboptimal way. More efficient approaches are desirable.

Recently developed algorithms provide efficient curve rendering with little CPU overhead and a significant reduction in vertex and polygon count. Some issues remain before the approach can be used for rendering in an OpenVG implementation.

This thesis builds on these algorithms to develop an approach that can be used for a conformant OpenVG implementation. A number of issues, mainly related to precision, robustness and missing features, are identified. Solutions are suggested and either implemented in a prototype or left as future work.

Preliminary tests compare the new approach to traditional approximation with line segments.

Vertex and triangle count as well as the simulated tile list counts are lowered significantly and CPU overhead from subdivision is avoided or greatly reduced in many common cases. CPU overhead from tessellation is eliminated through the use of an improved stencil buffer technique.

Data-sets with different properties show varying amounts of improvement from the new approach. For some data-sets, vertex and triangle count is lowered by up to 70% and subdivision is completely avoided, while for others there is no improvement.

Contents

List of Figures

List of Tables

Glossary

Alpha Value	Value used to control color blending. Often used to give the illusion of transparency, where an alpha value of 0 means fully transparent and an alpha value of 1 means fully opaque.
Anti-Aliasing	A collection of methods that reduce the jagged appearance of diagonal lines and primitive edges. This jagged appearance can easily be seen in figure 2.6. See supersampling and multisampling.
Attribute	See vertex attribute.
AGP	Accelerated Graphics Port. A special bus in PCs only used for graphics cards.
API	Application Programming Interface. A set of procedures and functions, also called programming library.
Baseline	The line between the starting point and the end point of the segment.
Binning	See tiling.
Bitmap	Strictly speaking this means a 2D array of bits, but it is often used as a synonym for pixmap.
Blend Mode	A setting of the color buffer blend unit. The color of a primitive can for example be added or subtracted from the render target.
Color Quantization	Process of reducing the number of unique colors used in an image.
Control Polygon	Polygon formed by the starting point, control point(s) and end point of a Bézier curve.
Culling	A process that removes invisible primitives from the pipeline.
Deferred Rendering	An approach to rendering where draw-calls are not executed as soon as they are issued by the application. They are instead buffered, and rendered one frame later. This makes it possible to perform some extra optimizations. See tile based rendering.
Depth Buffer	A buffer attached to the render target which stores the distance from the camera to each pixel.
Direct3D	A 3D graphics API by Microsoft.
Dithering	Process of giving the illusion of better color resolution by adding noise before color quantization. See color quantization.
Downsampling	The process of reducing the resolution of a pixmap. For example this can be done by taking the average of each 2x2 pixel quad in the pixmap.
Draw-Call	Batch of primitives to be drawn, sharing render states. Draw-calls are issued from the application to the driver.
Fill-Rate	The speed at which the GPU can draw pixels. Modern GPUs have very high fill-rate. See overdraw.
Fill Rule	A rule that specifies what is the inside and outside of a shape that self-intersects.
Fixed-Function Pipeline	A type of GPU pipeline where the operations performed on fragments/pixels and vertices can only be configured to a limited degree. See programmable pipeline.

Fragment	An element on which the fragment shader is executed. There are one or more fragments per pixel. The words fragment and pixel are used interchangeably in this thesis. See pixel.
Fragment Shader	A program (or a processor) that computes the color of a fragment. See fragment and shader.
Frame	A single rendered image in an animated sequence. For all our practical concerns, each frame is a separate render target.
Frame Buffer	The render target that is displayed on the screen.
Frustum Culling	The task of removing geometry that is not visible in the camera's field of view.
GPU	Graphics Processing Unit. A dedicated graphics rendering device.
Immediate Mode Rendering	An approach to rendering where the GPU starts rendering primitives as soon as they are issued by the application. See deferred rendering.
Index Buffer	A list of vertex indices. Often represents the topology of one physical object. Defines which vertices in a vertex buffer that compose each primitive.
Interior Polygon	The polygon formed by the start/end points of all segments in a path.
Kernel Mode	The CPU execution mode in which device-drivers run. In this mode, the CPU can access all physical-memory addresses, and all hardware.
Khronos Group	The standardization board responsible for creating among other things the OpenGL ES and OpenVG API specifications.
Kokojima et al's Approach	A variant of Loop and Blinn's approach where the stencil algorithm is used for rendering the interior polygon. See Loop and Blinn's approach.
Loop and Blinn's Approach	A path rasterization technique where a coarse bounding polygon is rendered around each segment and the fragment shader is used to discard pixels that should not be drawn. The original version then renders the interior polygon using Delaunay tessellation.
Mali series GPUs	Series of tile based GPUs developed by ARM. Mali200/GP2 is a programmable solution while Mali55 is fixed-function.
Multisampling	An approach to anti-aliasing. Rasterization is done at higher resolution than the render target, and then downsampled. Fragment shading is still done at the resolution of the render target.
OpenGL	Open Graphics Library. A standard graphics API based on IrisGL developed by SGI.
OpenGL ES	A version of OpenGL targeted at embedded systems.
OpenVG	Open Vector Graphics. A standard API for rendering 2d vector graphics.
Overdraw	The average number of times each pixel in the render target is modified during rendering of a single frame. Although high overdraw means that redundant rendering is performed, this is often the cheapest alternative. See fill-rate.
Path	A flexible type of graphic primitive that consists of multiple subpaths. See subpath.
PCI	Peripheral Component Interconnect. A bus in PCs used to connect extension cards into the system.
Pixel	A picture element. Consists of color and sometimes also alpha, depth and stencil information. The words fragment and pixel are used interchangeably in this thesis.
Pixel Shader	See fragment shader.
Pixel Units	See surface coordinates.
Pixmap	A 1D, 2D or 3D array of pixels.

Polygon	A closed shape consisting of connected line segments. Can be represented by a list of vertices.
Position and Varyings	The output values from a vertex shader. *Position* refers to the vertex' 2D render target position and is a required output. Varyings are optional output values that are later taken as input to the fragment shader. See shaded vertices.
PowerVR MBX	Fixed-function handheld GPU from Imagination Technologies.
PowerVR SGX	Programmable handheld GPU from Imagination Technologies.
Primitive	In the context of GPUs: An object that a GPU can render natively: A point, a line or a triangle. In the context of vector graphics: A geometric shape such as a rectangle, polygon or path. See polygon and path.
Programmable Pipeline	A type of GPU pipeline where the operations performed on fragments/pixels and vertices can be programmed in a flexible programming language. See fixed-function pipeline.
Rasterization	The process of finding the fragments that are covered by a primitive.
Recursive Subdivision	An algorithm that recursively subdivides curved segments until they can be approximated by lines (or something else that can be rasterized directly.)
Render State	Collection of states that defines how primitives will look when rendered.
Render Target	A pixmap that is rendered to by the GPU. It can also contain a depth buffer and a stencil buffer. Can be the frame buffer or a texture.
Rendering	The process of generating an image by a computer system.
Resampling	The process of changing the dimensions of an image. This is done by sampling into the original resolution image with filtering. See downsampling.
Segment	Paths are defined using an array of segments, each representing a piece of its outline. Examples: Line, quadratic curve, elliptical arc.
Sliver	Triangle with two long and one short edge. Undesirable for performance reasons on most GPUs.
Surface	See render target.
Surface Coordinates	Coordinate system where the axes are aligned with the pixels of the render target and one unit corresponds to the width and height of a pixel. Same as pixel units.
Shaded Vertices	A term used to collectively refer to the output data from the vertex shader: Position and varyings.
Shader	Two meanings: 1. A program that is executed in the GPU. 2. A unit in the GPU that executes such a program. Shaders were originally used for light calculations, but in new hardware they can be used for a wide range of computations. Because of this evolution, the term *shader* is now misleading, but still used. See fragment shader and vertex shader.
SoC	System on a Chip. A system where all components are integrated in one chip.
Stencil Algorithm	An algorithm for filling polygons with little CPU overhead.
Stencil Buffer	A buffer attached to the render target which stores an integer for each pixel. The application controls how this extra piece of information is used.
Subpath	A series of connected segments such as Bézier curves, elliptical arcs and lines. Multiple subpaths make up one path. See path.
Supersampling	Brute force approach to anti-aliasing. Rendering is done at higher resolution than the render target, and then scaled down with filtering.

Tessellation	Splitting a polygon into an equivalent set of polygons. In this report, tessellation and triangulation is used unterchangeably. See triangulation.
Texel	Texture element. A pixel which is part of a texture. See pixel.
Texture	A pixmap used as input to a fragment shader. The fragment shader can sample the texture at any desired coordinate. If desired, the result can be filtered using derivatives of the sampling coordinates.
Texture Coordinates	The fixed-function equivalent of vertex attributes. See vertex attribute.
Tile	A square of pixels that is a part of the render target. Tile based renderers render one tile at a time, making it possible to cache this tile in the core.
Tile List	A list of the rendering commands to be performed on a tile. See tile list command.
Tile List Set	Tile lists for all tiles in a render target.
Tile List Command	A single command in the tile list. The most important command is the Primitive command. There may also be some control flow commands but this depends on the implementation.
Tile Based Rendering	A form of Deferred Rendering where the render target is divided into fixed-size tiles. The tiles are rendered one by one. This makes it possible to efficiently buffer intermediate values of the render target.
Tiling	Calculating which tiles are covered by a primitive, and adding that primitive to the corresponding tile lists.
Time-To-Market	The amount of time it takes to get a product from idea to marketplace.
Triangulation	Splitting a polygon into an equivalent set of triangles. See tessellation.
T&L	Transform and lighting. Used as a synonym to per vertex operations. On programmable GPUs, the vertex shader often performs other tasks than transform and lighting, so the term is somewhat outdated.
Uniform	A value that is constant for a whole draw-call. It can be used both in vertex shaders and fragment shaders.
User Mode	The CPU execution mode in which applications run. In this mode, each application has its own address space, and can not access the memory of other applications.
Varying	A value that is an output from the vertex shader. It is linearly interpolated across a primitive by the rasterizer, and then taken as input to the fragment shader. Since vertex shaders are not used in this thesis, varyings are equal to vertex attributes.
Vertex	A point in 3D space with associated attributes. Vertices are used to define primitives. See primitive.
Vertex Attribute	Value specified per vertex and taken as input to the vertex shader. Also called texture coordinates. Since vertex shaders are not used in this thesis, attributes are passed through the vertex shader stage and become varyings. See texture coordinates and varyings.
Vertex Buffer	A list of vertices. Often contains vertices of one physical object.
Vertex Shader	A program (or a processor) that takes a vertex buffer as input, and produces transformed vertex positions and varyings as output. See shader and shaded vertices.
Z-buffer	See depth buffer.

Interpretation of Assignment Text

The main task defined by the project assignment is to describe an algorithm for rasterization of filled OpenVG paths using the GPU. The feature sets of OpenGL ES 1.x and 2.0 represent two typical hardware setups for handheld devices that the work should be targeted at. In addition, four optional tasks are defined that can be performed if time permits.

The main task is pretty clearly defined in the assignment text: "Describe an algorithm for rendering the interior of paths efficiently, with support for all possible fill techniques, using an OpenGL ES 2.0 conformant GPU, with possible hardware additions and modifications." I believe that "all possible fill techniques" refers to the two fill rules defined by OpenVG, and that it should be possible to support various paints and blend modes in the future. Possible hardware additions and modifications refer to simple things like more stencil buffer bits or making specific assumptions about rasterization rules. Larger and more involved changes to the GPU architecture is not an option in this case.

However, a requirement specification that includes all the optional tasks is given below. It defines the goal of the work in the long term rather than a bare minimum that must be accomplished in this thesis.

The requirement specification for the prototype:

1. Show a significant improvement over traditional polygonal approximation and tessellation methods.
2. Support (efficiently) both fixed-function and programmable GPUs. (OpenGL ES 1.x and 2.0 feature sets.)
3. No need for my prototype to support all paints and blend modes, but this must be implementable at a later time.
4. Algorithms must be robust. Should be able to rasterize all paths within the requirements of the OpenVG specification.
5. Support stroking and filling with both fill rules.

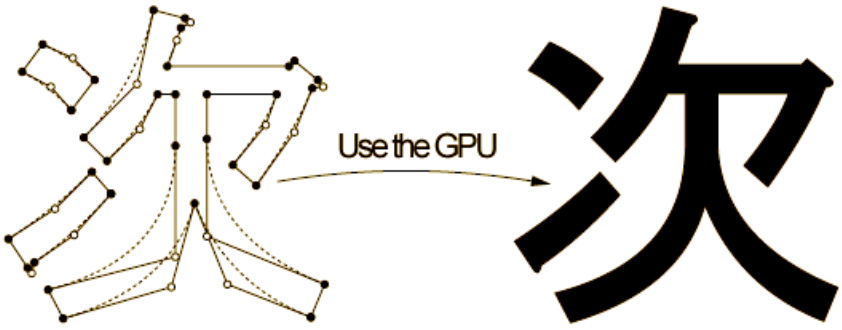


Figure 1: Illustration of the assignment goal. Drawing modified from [32].

Figure 1 illustrates the goal of the assignment.

According to the assignment, algorithms that take advantage of GPU features should be used. This eliminates many classic path rasterizing algorithms. (See chapter 2.1.)

It is assumed that the final implementation does not necessarily work through OpenGL ES, but can be operating the hardware at a low level of abstraction. In addition, minor modifications and additional features to the GPU hardware can be assumed.

The algorithms should be implemented and tested, but not necessarily run on a handheld device. Improvements over a traditional approach should be measured, especially improvements related to long, thin triangles.

1 Introduction

While traditional *bitmap* graphics represent images at a fixed resolution using a grid of color values, vector graphics define shapes using smooth, resolution-independent primitives. Vector graphics are used for user interfaces, illustrations, font rendering and more in a wide range of applications.

During the last few years, handheld devices have become increasingly powerful and feature-rich. It is expected that an increasing number of devices will contain dedicated GPUs (graphics processing units) capable of high quality 3d graphics for games. It is of interest to use the same hardware for accelerating vector graphics.

OpenVG is a new API for vector graphics rendering on a wide range of devices from desktop to handheld. Graphical shapes, or *paths*, are defined using a sequence of segments such as Bézier curves and lines. Implementations can use different algorithms and ways of accelerating the rendering process in hardware transparent from the user application.

This project aims to create an efficient path rasterizer that can be used in an OpenVG implementation.

OpenVG has clearly defined criterions for how much error is allowed in rasterization of paths. A conformance test suite is under development, and will be used to test OpenVG conformancy of commercial implementations. It is however not freely available and is therefore not used for this project.

The number of vertices and triangles per path should not be too high, as new geometry data must be transferred from the CPU to GPU memory before rendering.

The shape of the triangles also affects memory traffic. In tile based renderers, tile list commands must be stored in memory and read back. In immediate mode renderers, cacheability of render target contents is affected. Long, thin triangles generate excessive memory traffic and are therefore undesirable.

Fill-rate is not expected to be a problem for vector graphics. GPUs are optimized for high amounts of overdraw. A typical handheld configuration can fill the screen with pixels around 1.000 times per second. The CPU should however be used sparingly to free up time for user applications.

State of the art vector graphics solutions perform much processing in the CPU, involve a large amount of traffic from the CPU to the GPU, and generally use the GPU in a suboptimal way. More efficient approaches are desirable.

Recently developed algorithms provide efficient curve rendering on the GPU with low polygon and vertex numbers and little CPU overhead. Some issues remain before the approach can be used for rendering in an OpenVG implementation. I have found evidence of only one partial implementation of these techniques.

This project builds on these algorithms to develop an approach that can be used for a conformant OpenVG implementation. A number of issues, mainly related to precision, robustness and missing features, are identified. Solutions are suggested and either implemented in a prototype or left as future work.

Methods for efficient rendering of quadratic curves and elliptical arcs on both fixed-function and programmable GPUs, and cubic curves on programmable GPUs are developed.

Cubic curve rendering involves a significant amount of floating point calculation on the CPU, needs high precision in the GPU to perform well, and can not be used on fixed-function GPUs. Approxima-

tion with quadratic curves is however feasible, and may be more visually appealing than a polygonal approximation with the same error threshold.

Preliminary tests compare the new approach to traditional approximation with line segments.

Vertex and triangle count as well as the simulated tile list count is lowered by up to 70%. However, in some benchmarks there is no measurable improvement.

The largest improvement in vertex and triangle count is shown for big, smoothly curved shapes. There is little improvement for detailed graphics such as Chinese text. The reason is that small details can be easily approximated without introducing much error. Big curves are however not easily approximated by lines, so that is where the new technique excels.

Subdivision on the CPU is usually avoided or greatly reduced, provided the GPU can rasterize segments with sufficient precision. Cubic curves must often be subdivided on GPUs that support only OpenGL ES' minimum precision requirement.

CPU-based tessellation is eliminated at the cost of some overdraw by using a variant of the stencil algorithm. GPUs are optimized for high fill-rate, so fill-rate is not expected to become a bottleneck.

1.1 Report Structure

This chapter describes the structure of the report and gives a brief description of the contents of the various chapters.

Note that the structure of the document does not reflect the working process. That is, I did not go through isolated phases of evaluating existing approaches, improving the algorithms and then finally implementing it. These processes were all performed in parallel, and there were of course also attempts at creating new algorithms as well as improving existing ones.

In addition to textual explanations and high-level pseudocode, object-oriented C++-like pseudocode is much used throughout the report when explaining various approaches and algorithms. The reader is expected to be familiar with C++ or a similar language, such as Java. Arrays are dynamic. For example, Segment[] can be interpreted as the C++ type std::vector. I also assume that there is garbage collection, so I will not delete objects explicitly.

The document is divided into the following main parts:

1 Introduction
The current chapter which contains an introduction, report structure and acknowledgements.

2 Background
This chapter gives relevant background information and goes through the necessary theory for understanding the work presented in this thesis. It will be referred to and used when developing the OpenVG rasterizer in later chapters. Unlike in later chapters, the theory and ideas presented in this chapter are not my own. (Later chapters will have clear references when presenting ideas that are not my own.)

3 State of the Art
Existing vector graphics software has been investigated to find which rasterization techniques are in use. They are evaluated and discussed in this chapter.

4 Evaluation of Algorithms for Path (and Polygon) Rasterization
This chapter aims to find the most promising approach to path rasterization for our purposes. The algorithms presented in the background chapter will be evaluated and compared. Since all of the considered path rasterization techniques also involve polygon rasterization, different approaches to polygon rasterization are also evaluated. The focus is on efficiency. Relevant statistics for comparing algorithms are also presented and discussed.

5 Novel Approaches and Improvements to Algorithms
The previous chapter concludes that Loop and Blinn's approach combined with the stencil algorithm

is the most suitable solution to our problem. Several issues must however be solved before it can be used for a conformant OpenVG implementation. These issues are solved in this chapter, and some new techniques that may improve performance are presented.

6 Path Rasterizer Architecture and Prototype Implementation
This chapter describes the new, efficient approach to OpenVG path rasterization and the implementation of the prototype. It ends with a summary of what has been accomplished so far in the thesis.

7 Prototype Verification
The prototype implementation verification process is described in this chapter. Although verification was specified as optional in the project assignment text, functional verification is performed, while verification of rasterization error is left for future work.

8 Benchmark Results and Discussion
This chapter describes how statistics from the prototype are collected for both the traditional polygonal approximation approach and the new, efficient path rasterization approach. The results are compared and discussed. The actual numbers from the collected statistics can be found in appendix B.

9 Conclusions
The conclusions of the thesis are given in this chapter.

10 Future Work
Ideas for tasks that are suitable for future work are presented in this chapter. Some techniques that were not complete enough to be included in earlier chapters are also described here.

The following appendices are included:

A Prototype User Manual
A user manual for the prototype implementation is provided in this appendix.

B Benchmark Results
Benchmark statistics collected by running the various test-cases under various configurations. Images of the tests are also provided.

C Source Code Reference Manual
Source code reference manual created using Doxygen.

1.2 Acknowledgements

I would like to thank Espen Åmodt and my technical supervisor Thomas Austad for valuable discussions and clarifications regarding the OpenVG standard and technical issues, and for being enthusiastic about this project. Morten Hartmann, my supervisor at NTNU, for early reading and feedback, and discussions regarding writing and technical issues. Mario Blazevic for help with issues related to intellectual property and with defining the assignment text.

2 Background

This chapter gives background information and theory that is used and referred to in later chapters.

An overview of vector graphics and their applications is given in chapter 2.1. It is followed by an introduction to hardware acceleration of computer graphics in chapter 2.2.

A typical driver stack of a GPU is described in chapter 2.3, followed by an introduction of the programming model for fixed-function and programmable GPUs in chapter 2.4. References are made to OpenGL ES to show how the API corresponds to the programming model.

The OpenVG API is described in chapter 2.5.

Chapter 2.6 describes two types of GPU architectures that are on the market today, and explains why they have different performance characteristics.

Various approaches to polygon rasterization in the GPU are described in chapter 2.7. Chapter 2.8 explains *recursive subdivision*, a much-used algorithm for creating polygonal approximations of smooth shapes.

The concept of *offset curves* is explained in 2.9. The chapter includes references to algorithms that could be suitable for future work related to my assignment.

Chapter 2.10 explains Loop and Blinn's recent approach for filling cubic and quadratic curve segments. Kokojima et al's approach of combining their curve rasterization algorithm with the stencil algorithm is also presented.

Most of chapters 2.2.1, 2.2.2, 2.3.2, 2.4, 2.6.1, and 2.6.2 have been adapted from [35], a project report written by me and Edvard Fielding. It is noted where applicable which parts are from there and which parts are new.

2.1 Vector Graphics

While *bitmapped* graphics represent images at a fixed resolution using a grid of color values, vector graphics define shapes using smooth, resolution-independent primitives.

Two-dimensional vector graphics are used for user interfaces, illustrations, fonts, CAD/CAM programs and more. They are used for scale-independent graphics in drawing applications such as Adobe Flash and Adobe Illustrator, for typesetting and illustrations in PDF and Postscript [6], for defining characters in font formats such as TrueType [32], and are used in CAD programs to design among other things aircrafts and cars [27].

New, flexible vector graphic libraries commonly use paths, a flexible type of primitive that consists of a series of connected segments. A single path can contain multiple types of segments such as quadratic curves, cubic curves and elliptical arcs. Typically, the interior as well as the outline (stroke) of each path can be rasterized and drawn using a variety of colors, textures and patterns, and transparency effects are often possible.

A number of algorithms have been developed for rasterizing vector graphics during several decades. Classic path rasterizing algorithms include the *midpoint algorithm* for drawing elliptical arcs, the *active*

edge list algorithm for filling polygons and more. These algorithms are discussed in [24].

Unlike CPUs, GPUs are optimized for massively parallel tasks and operations typical of graphics applications such as vector arithmetic. A large number of algorithms for curve rendering were created before GPUs became mainstream. New algorithms that take advantage of GPU features are desired. This makes most of the classic algorithms unsuitable for our purposes.

2.2 Hardware Accelerated Graphics

A simple approach to raster graphics is to have a representation of the display contents in a memory area called the frame buffer, each word corresponding to a pixel, and let the CPU directly manipulate the contents. Applications can then do their graphics tasks in the CPU and write the results directly to the frame buffer.

In the 1980 and early 90s, graphics adapters included additional functionality for simple manipulation and copying of frame buffer contents. These adapters sped up applications such as games and user interfaces, and were the precursors to modern GPUs [14].

2.2.1 Introduction of GPUs to the Consumer Market

This chapter has been adapted from [35]. It is originally based on [38] and [21] unless otherwise noted. The last two paragraphs are new for this thesis.

Real-time 3D graphics became common in personal computer and console games in the 1990s. This led to a great demand for hardware-accelerated graphics since it could provide higher resolutions and better image quality to games. The first 3D accelerators were little more than rasterizers that could draw primitives such as lines and triangles into the render target. Soon the accelerators were extended to be able to draw transparent and textured primitives.

At this time there was no real standard API. OpenGL was mostly used for professional applications, and the early versions of Direct3D (Microsoft's 3D graphics API) were extremely difficult to use. This meant that the manufacturers of 3D hardware made their own APIs. Examples of this are 3dfx Glide and Rendition Redline. A graphics card often only supported one of these APIs, so game programmers had to make different versions of the games to support the most common hardware. This gradually improved as Direct3D 5.0 was a lot simpler, and OpenGL became widely used on consumer 3D hardware.

The first card that could do transform and lighting (T&L) in hardware was introduced in 1999. The NVIDIA GeForce 256 offloaded the CPU and thus made far more complex model geometry possible. The term Graphics Processing Unit (GPU) was in fact introduced with this graphics card [9].

In 2000 Microsoft introduced Direct3D 8.0. It was the first API that supported programmable vertex and fragment shaders. This feature made the graphics hardware programmable, and that enabled far better image quality and many new special effects. The first card with this feature was the NVIDIA GeForce 3 (2001). Vertex and fragment shaders have been enhanced over the years, and are now both floating-point vector processors that can be programmed in a C-like language. This has made shaders easy to use.

I make a clear distinction between *fixed-function* and *programmable* GPUs. Fixed-function GPUs do not have general programmable vertex and pixel processors. Only a limited set of operations can be performed on pixels and vertices. Section 2.4.2 explains this difference in detail.

The GPU is much faster than the CPU in many tasks that are common in computer graphics, and are therefore no longer used only for traditional 3d graphics applications. For example, new desktop computer operating systems such as MacOS X and Windows Vista render the graphical user interface using the GPU.

2.2.2 Handheld GPUs

This chapter has been adapted from the chapter "Handheld Graphics" in [35], with some additions. It is originally based on [41] unless otherwise noted.

One of the first uses of handheld graphics was for graphical user interfaces. Early mobile phones had character displays and could not show raster graphics. These devices had utilitarian, text-based user interfaces.

Handheld game consoles with raster displays have been available at least since 1989 [7]. Handheld units with 3D graphics accelerator hardware were shipped in 2004. Among these was the Nintendo DS [8].

GPUs are now used in handheld game consoles, mobile phones as well as other handheld devices [42]. Most handheld devices should be cheap and small. For these reasons, it is desirable to use the same hardware for rendering graphical user interfaces and other 2d graphics as well as 3d graphics.

The feature set of new handheld GPUs such as ARM's Mali 200 matches the feature set of desktop and stationary game consoles, but the speed is lower. However, since the display resolution of handheld devices is normally much lower than for these systems, the perceived performance can be similar.

2.3 GPU Driver Stack

This chapter is originally based on [5], [43] and [45].

Support for the most common APIs must be implemented in the GPU's software, and a device-driver is needed for communication with the hardware.

2.3.1 The Khronos Group

The website for the Khronos Group has the following description of their organization: "The Khronos Group is a member-funded industry consortium focused on the creation of open standard, royalty-free APIs to enable the authoring and accelerated playback of dynamic media on a wide variety of platforms and devices. All Khronos members are able to contribute to the development of Khronos API specifications, are empowered to vote at various stages before public deployment, and are able to accelerate the delivery of their cutting-edge 3D platforms and applications through early access to specification drafts and conformance tests." [11]

The Khronos Group's main activity is to create specifications for APIs that enable applications to interface with hardware from multiple vendors without having a separate code path for each. Their APIs are mainly targeted, but are not limited to handheld devices such as mobile phones. Their specifications include OpenGL ES 1.x and 2.0 and OpenVG.

2.3.2 Software Driver Structure

Most of this chapter has been adapted from [35].

The software package that is supplied with the GPU is usually divided into (at least) two layers; A front-end and a device-driver. The application communicates with the front-end using a standardized API such as OpenGL or OpenVG. The front-end uses operating system functionality to communicate with the device-driver, and the device-driver communicates with the GPU hardware. Figure 2.1 shows a system with an application, a front-end implementing the OpenVG API, a device-driver, GPU hardware and a display.

The reason for separating the front-end and the device-driver into two layers is that modern operating systems have two running modes: User and kernel. User mode applications are protected, and can not access hardware directly. This prevents them from disturbing other applications or crashing the system. Programs running in kernel mode have full access to all memory addresses and all hardware. They can crash the computer so that only a reboot will restore normal operation. It is therefore extremely important

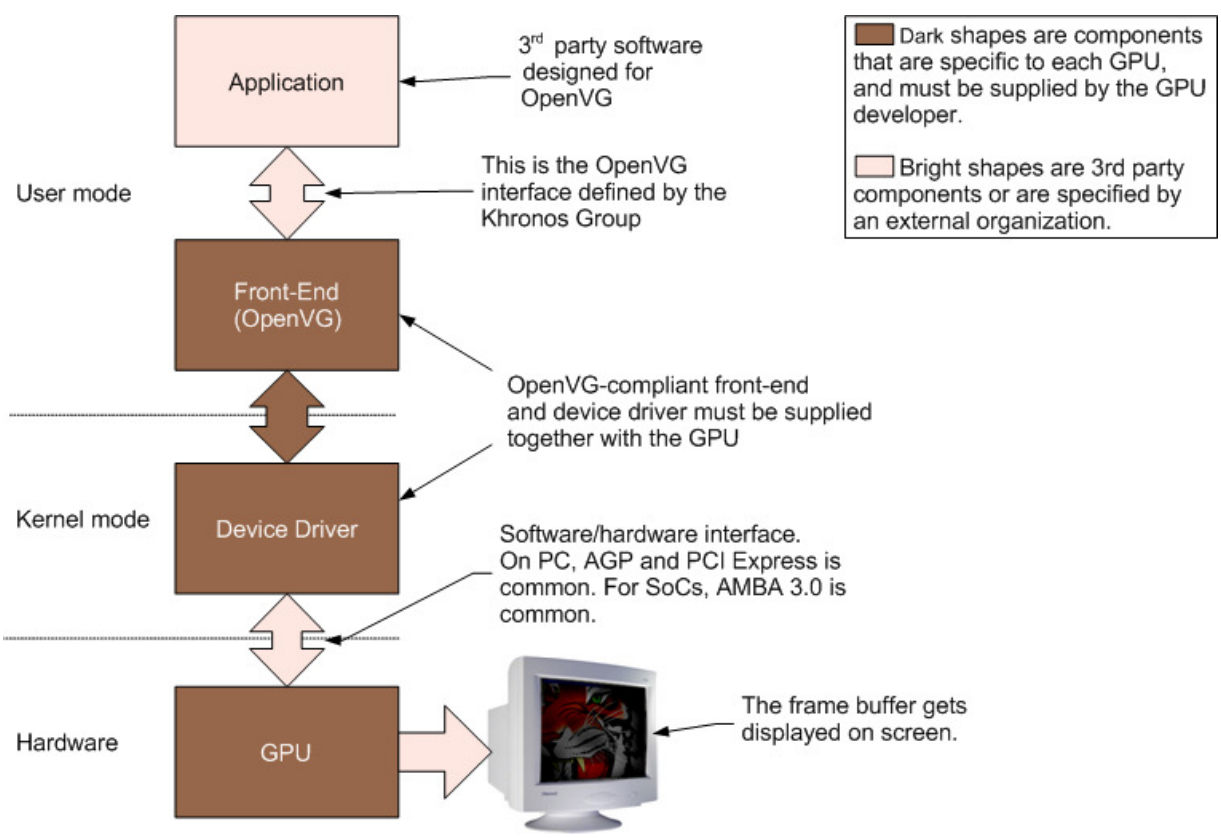


Figure 2.1: The layers and interfaces of an OpenVG setup.

that the device-driver is stable and does not contain any bugs. In addition, software running in kernel mode is very difficult to debug. It is therefore common to make the device-driver as small as possible. Most of the desired functionality is therefore implemented in the front-end driver.

In a system that implements multiple APIs, for example OpenVG and OpenGL, many different arrangements are possible. Another possibility is to implement OpenVG on top of OpenGL ES, using OpenGL ES extensions when additional functionality is required. In a highly optimized solution, OpenGL ES and OpenVG are likely to have separate front-ends, with a shared base driver in kernel mode.

2.4 GPU Programming Model

Most of this chapter has been adapted from [35]. Explanation of OpenGL ES 1.x and fixed-function functionality has been added. The chapter is originally based on [5], [43] and [45].

This chapter gives an overview of the programming model of GPUs through the OpenGL ES 1.x and 2.0 APIs. OpenGL ES exposes the graphics systems' functionality to the application in the form of a *conceptual pipeline* which is modelled after how most GPUs work.

2.4.1 The OpenGL ES APIs

OpenGL ES (GLES) 1.x and 2.0 are APIs for rendering 3D graphics, created by the Khronos Group. (See chapter 2.3.1) They are based on OpenGL with the goal of reducing the complexity of that API. Unlike OpenGL, GLES does not include redundant functionality for old and redundant features. This has been made possible by dropping compatibility with OpenGL. Modern GPUs typically emulate obsolete features of older APIs such as OpenGL by using features available in OpenGL ES.

While OpenGL includes functionality for both fixed-function and programmable GPUs in a single API, the Khronos group have chosen to split the API in two: GLES 1.x for fixed-function GPUs and GLES 2.0 for programmable GPUs, as shown in figure 2.2 A vendor can choose to support only one or both of

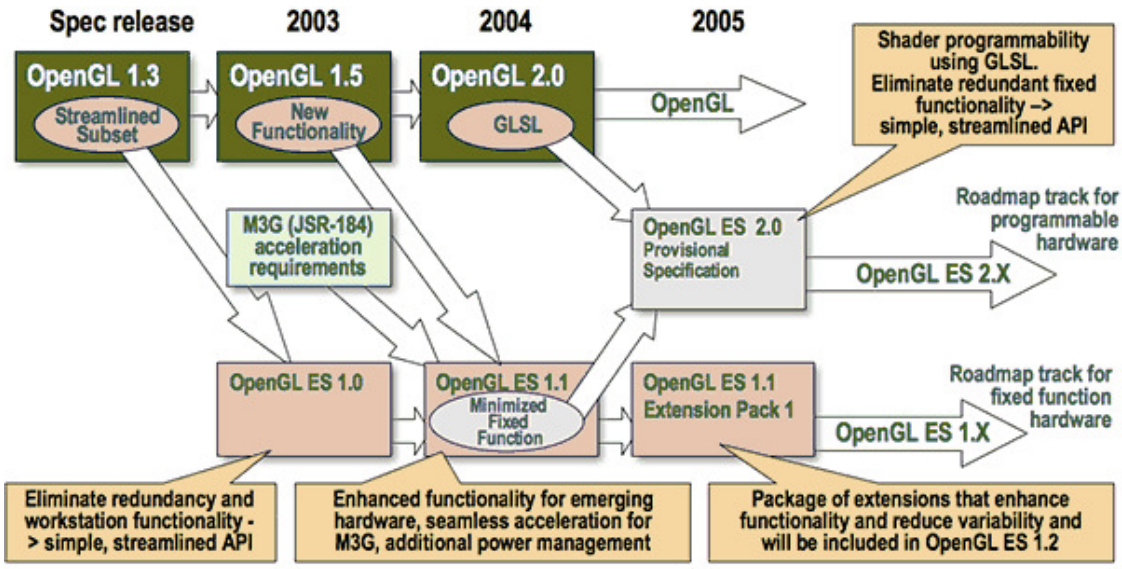


Figure 2.2: OpenGL ES roadmap - two tracks [11].

the standards.

I choose to use OpenGL ES as a real-world example when I explain the different stages of the graphics pipeline in chapter 2.4.2. Its simplicity and flexibility as well as a close mapping to modern hardware makes it a suitable example.

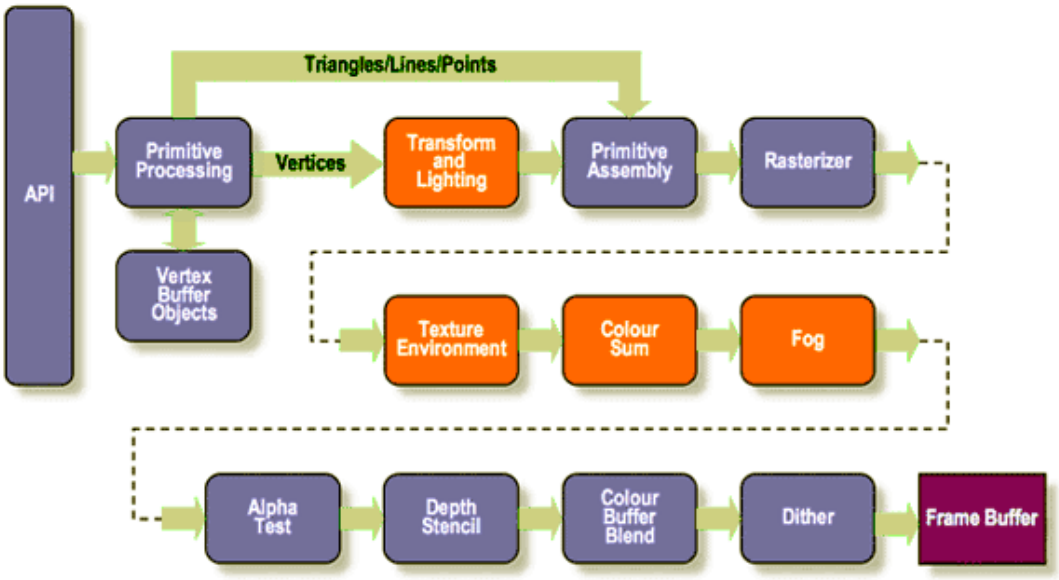


Figure 2.3: OpenGL ES 1.x fixed-function pipeline [11].

The OpenGL ES 1.x pipeline is shown in figure 2.3, and the OpenGL ES 2.0 pipeline is shown in figure 2.4. They will be explained in chapter 2.4.2 using a simplified diagram.

Most of the capabilities of a modern industry-standard handheld GPU is exposed in either GLES 1.x or GLES 2.0. It may however have some additional features that are exposed through other APIs such as OpenVG.

2.4.2 Pipeline Walkthrough

The programmer's view of the Open GL ES graphics system is that of a pipeline with several stages. Each stage is a process which takes its input from the previous stage, performs some processing, and sends its output to the next stage.

The output from the end of the pipeline is written to the *render target*. The render target is a two-dimensional map of pixels - a *pixmap*. The start of the pipeline takes *primitives* as input. A primitive

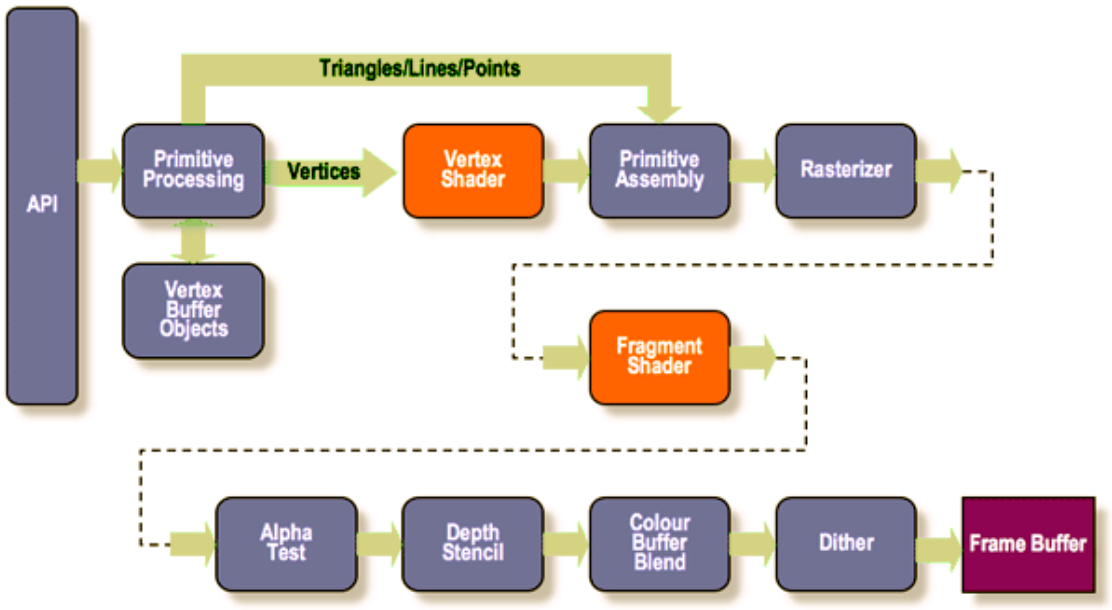


Figure 2.4: OpenGL ES 2.0 programmable pipeline [11].

is a simple geometric figure. Most graphics systems support points, lines and triangles. Primitives are specified using *vertices* and *indices*. A *Vertex Buffer* specifies positions and properties for several points in space. The *index buffer* defines primitives by referring to the vertices in the vertex buffer. Points, lines and triangles are defined using one, two or three vertices respectively.

Most of the pipeline's stages are *fixed-function*, while others are *programmable*. Fixed-function stages have little flexibility in the process they perform. Programmable stages are actually general microprocessor cores that can perform any operation to their input and are free to generate its output in whatever way is desirable.

The application performs rendering by issuing *draw-calls* to the graphics system. A draw-call consists of a vertex buffer, an index buffer and a *render state*. The render state specifies what processes should be performed to the render objects at each stage of the pipeline and includes:

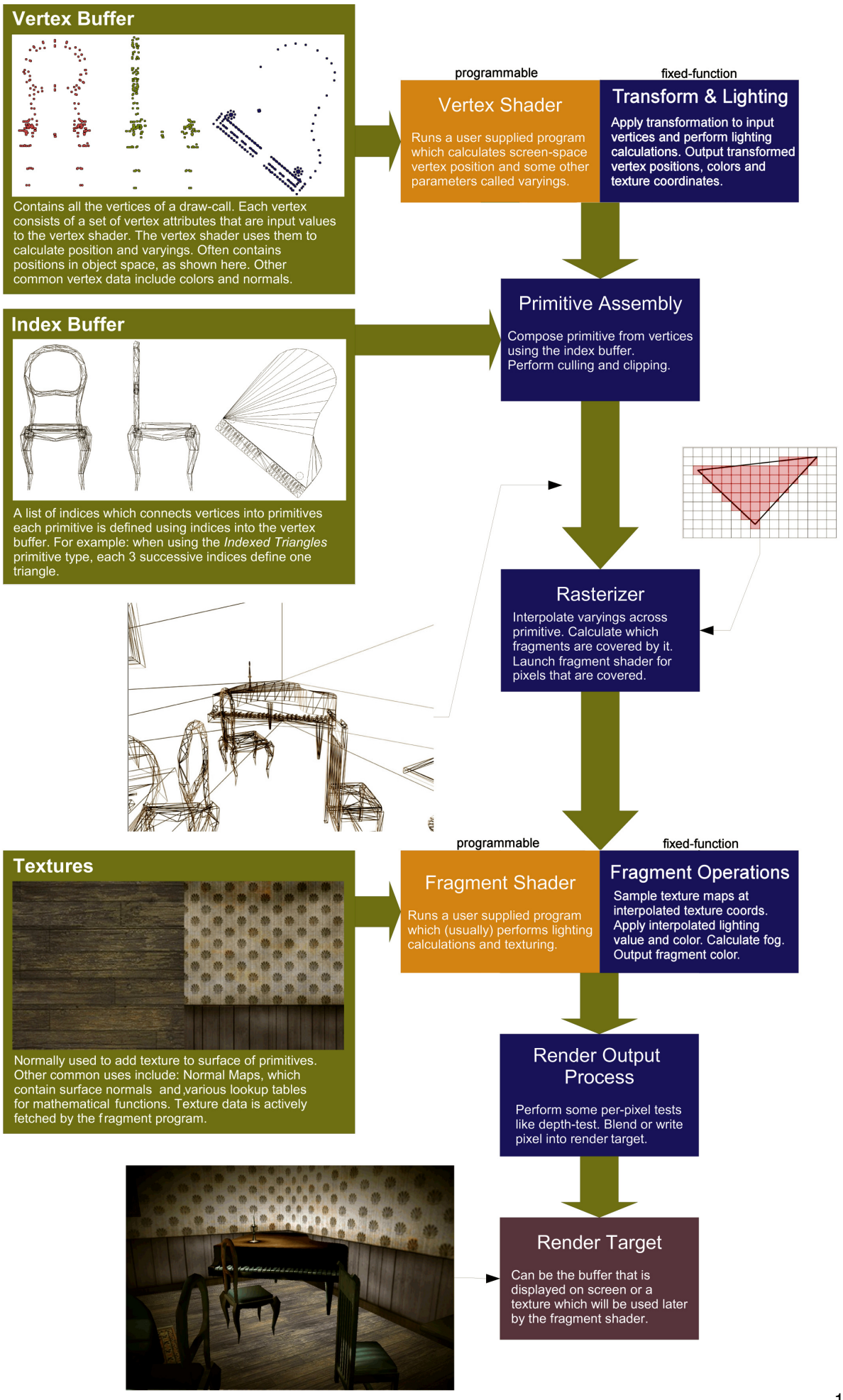
- Shader programs
- Uniforms
- Texture states
- Depth/stencil operation states
- Blend modes

Most of the fixed-function stages can be configured to set up their operations in some static, limited way, but they can not be programmed. For the programmable stages, the render state contains programs that are to be uploaded and executed. The programmable stages will execute these programs once for each input object they receive from the previous stage. The program should create an output object which will flow down to the next stage of the pipeline.

I make a clear distinction between *fixed-function* and *programmable* GPUs. Fixed-function GPUs do not have general programmable vertex and pixel processors. Only a limited set of operations can be performed on pixels and vertices. I have made figure 2.5 to illustrate a typical programmable API graphics pipeline. The colors of the diagram have the following meanings:

- Blue means fixed-function stage. Most of these stages can be configured in some way, but they can not be programmed.

Figure 2.5: Conceptual illustrated pipeline.



- Orange means that on a programmable GPU, this stage runs an application-specified shader program. These stages are typically implemented as vector processors.
- Green is used on arrows and on boxes which describe what kind of data flows through the pipeline.
- Purple means that this is a data structure stored in memory.

In a programmable GPU, the fragment and vertex operation stages are implemented as general, programmable microprocessors that run one thread per input. Using OpenGL ES 2.0 functionality, the application can upload the program that is to be run to generate the required output from the stage. In fixed-function GPUs, these stages are fixed units that perform pre-defined (configurable) operations. The operations are configured using OpenGL ES 1.x functionality.

Figures 2.3 and 2.4 show overviews of the OpenGL ES pipelines, as illustrated by the Khronos Group. They are similar to my illustration, but are a bit more detailed. The main difference is that some of the stages are expanded into multiple stages. I also think that they have a slightly confusing illustration of the inputs to the primitive assembly stage.

I will now go through the pipeline stages of figure 2.5 and explain them. Each stage will be related to the corresponding stages in the OpenGL ES pipeline.

Vertex Shader/Transform & Lighting (T&L)

This stage is different for fixed-function and programmable GPUs. On fixed-function GPUs, it has limited flexibility and can perform only pre-defined operation, while on programmable GPUs, it runs an application-defined program.

The main task of the Vertex Operations stage is to transform vertices from the object's own coordinate system to the render target's coordinate system. This includes translation, rotation, scaling and perspective projection. The Vertex Operations stage can also do other tasks such as per-vertex lighting calculation.

The output from this stage consists of position, and optionally a set of values called varyings that are to be interpolated across the primitives. The interpolated values are used as inputs to the fragment operations. For fixed-function GPUs these varyings are limited to a single color and a specified number of texture coordinates. On programmable GPUs varyings can be anything that the application wants to pass on to the fragment operations stage.

Fixed-Function GPUs (GLES 1.x): The vertex coordinates are transformed by application-defined matrices and lighting calculations are performed. A common term for the fixed-function vertex operations is T&L (Transform and Lighting). The GPU may also be able to do other things such as blending between different matrices. However, the operations are not generally programmable.

Programmable GPUs (GLES 2.0): The inputs to the vertex operations stage as well as the shader program itself are defined by the application. The shader can use constants (called uniforms) that remain the same for all vertices in a draw-call, and vertex attributes such as color, texture coordinates etc. that change from vertex to vertex. The vertex attributes are stored in lists called vertex buffers.

Primitive Assembly

The Primitive Assembly stage fetches shaded vertices from the vertex operations stage and assembles them into primitives. Finally, it performs back-face culling, frustum culling and clipping before the result is sent to the rasterizer stage.

Back-face culling is the process of discarding primitives which are facing away from the screen. Frustum culling is the process of discarding the primitives which are fully outside the screen. If a primitive is partly outside the screen, clipping removes these parts of the primitive so they are not sent to the rasterizer.

On many GPUs, the rasterizer can only draw triangles. If this is the case, it is also the job of the Primitive Assembly stage to convert other primitives to triangles. Points and lines are drawn as rectangles which

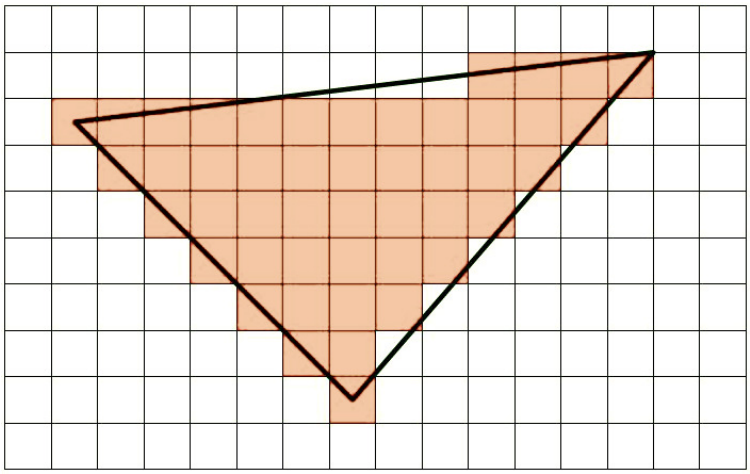


Figure 2.6: The fragments covered by a triangle, as found by a rasterizer.

are easily divided into two triangles.

The dataflow is illustrated a little differently in my illustration (Figure 2.5) and the OpenGL ES pipelines (figures 2.3 and 2.4). In my version, the primitive assembly stage reads the index buffer. It asks the vertex operations stage for the vertices it needs and then assembles the primitive. In the OpenGL ES pipelines however, there is a *Primitive Processing* stage before the vertex shader/T&L stage. It is this stage which reads the index buffer and tells the vertex shader/T&L which vertices to shade. It then informs the primitive assembly stage which type of primitive to assemble - point, line or triangle. When the vertices arrive to the primitive assembly stage it has the information it needs to assemble the primitive. My version and the OpenGL ES versions of the pipeline are functionally equivalent.

Rasterizer

The Rasterizer computes which of the fragments in the render target that are covered by a primitive. For a triangle, this can be done by setting up three line-functions for the edges, and testing that they all return a non-negative value for the coordinates of each fragment. The rasterizer also linearly interpolates varyings across each primitive using a weighted sum of the shaded values from all the primitive's vertices. Figure 2.6 shows the rasterizer output for a triangle.

Fragment Shader/Operations

When the Rasterizer finds a fragment that is inside a primitive, it is sent to the fragment operations stage.

This stage is different for fixed-function and programmable GPUs. On fixed-function GPUs, it has limited flexibility and can perform only pre-defined operations in a specific order, while on programmable GPUs, it runs an application-defined program.

As input the fragment operations stage takes uniforms, varyings and textures. The varyings come from the rasterizer. Textures and uniforms as well as the shader program itself are defined by the render state, which is issued by the application. The fragment operations stage samples textures, computes per-pixel lighting and performs any other operation needed to find the values of the fragment.

Render Output Process

The render output process inserts fragment values from the fragment shader into the render target. A number of tests and computations are performed. The application has a fairly high level of control of the processes in this stage, but it is not generally programmable.

The render output stage can be divided into 4 processes called *Alpha Test*, *Depth/Stencil Test*, *Color Buffer Blend* and *Dither*. These are represented as unique stages in the OpenGL ES pipeline illustrations (figures 2.3 and 2.4). The two representations are functionally equivalent.

The Alpha Test process performs a comparison of the alpha value of the fragment against a constant, and removes the fragment from the pipeline if this test fails.

The Depth/Stencil process compares the depth value and the stencil value of the fragment against the depth buffer and stencil buffer, and removes the fragment from the pipeline if any of these two tests fails.

The Color Buffer Blend unit mixes the color of the fragment with the current color of the render target pixmap. The blend operation can be set up to add, multiply, perform linear interpolation etc. The different setups of the color buffer blend unit are often called blend modes.

Figure 2.7: A grayscale image dithered for display on a monochrome screen.

Displays on handheld units often can often display a fairly low number of unique colors. Usually, the GPU operates in a higher color resolution than the display, and must therefore convert the frame buffer image to low color resolution before it can be displayed. This process is called *color quantization*. Dithering is a technique which can be used to reduce the perceived error from this process. It works by adding low-amplitude noise to the image before it is quantized. This is parallel to when a printer gives the illusion of gray by writing small black spots on a white background. Figure 2.7 illustrates this.

Render Target

The pipeline ends with the render target. This is the pixmap which contains the result of the rendering. The render target can be a texture, or it can be the frame buffer. When the application is done issuing draw-calls for a single frame, it can start rendering to a new render target. If the previous render target was the frame buffer, it will be displayed on screen. If it was a texture, it will now be available as inputs to the fragment shader.

The OpenGL ES pipeline illustrations call this the frame buffer. This is a bit inaccurate, as the render target does not have to be the frame buffer.

2.5 The OpenVG API

This chapter is based on information from the OpenVG Specification [6].

OpenVG is a new API for vector graphics rendering on a wide range of devices from desktop to handheld. It provides a vendor-independent interface for applications. Implementations can use different algorithms and ways of accelerating the rendering process in hardware.

It has a drawing model that is similar to, and can be used for implementing existing APIs and formats, including PostScript, PDF, Flash, Java2D, and SVG.

A *path* is a geometric shape that can be drawn using the OpenVG API. The geometry is defined by a sequence of *segment commands*. The interior as well as the outline (stroking) can be drawn in a variety of ways.

Each path consists of one or more *subpaths* - unconnected shapes that contribute to the path.

When rendering a path, it can be *filled* and/or *stroked* using a selected *paint* and *blend mode*. A 2d affine

transformation can be applied to the rendered geometry through an application defined user-to-surface matrix.

2.5.1 Paint and Blend Modes

Paints are used to choose which colors should be used when drawing a path. They can be single-colored, but more complicated options are available. For example, various forms of gradients as well as images can be used.

Blend modes are applied at the end of the OpenVG pipeline and define per pixel output color as a function of paint color and the color that is already in the frame buffer. They are mainly used for transparency effects.

2.5.2 Filling and Stroking

When a path is drawn, it can be either stroked, filled, or both.

Filling a path means that its inside/interiors are drawn using the desired paint and blend mode. A fill rule is required to decide which parts are defined as inside and outside the path, as explained in 2.8.

Stroking means to draw the outline of the path, as if the segments were stroked with a pen. There are a lot of options available to define how the stroke will look. This includes stroke width, dashing, end cap style and line join style.

2.5.3 Fill Rules

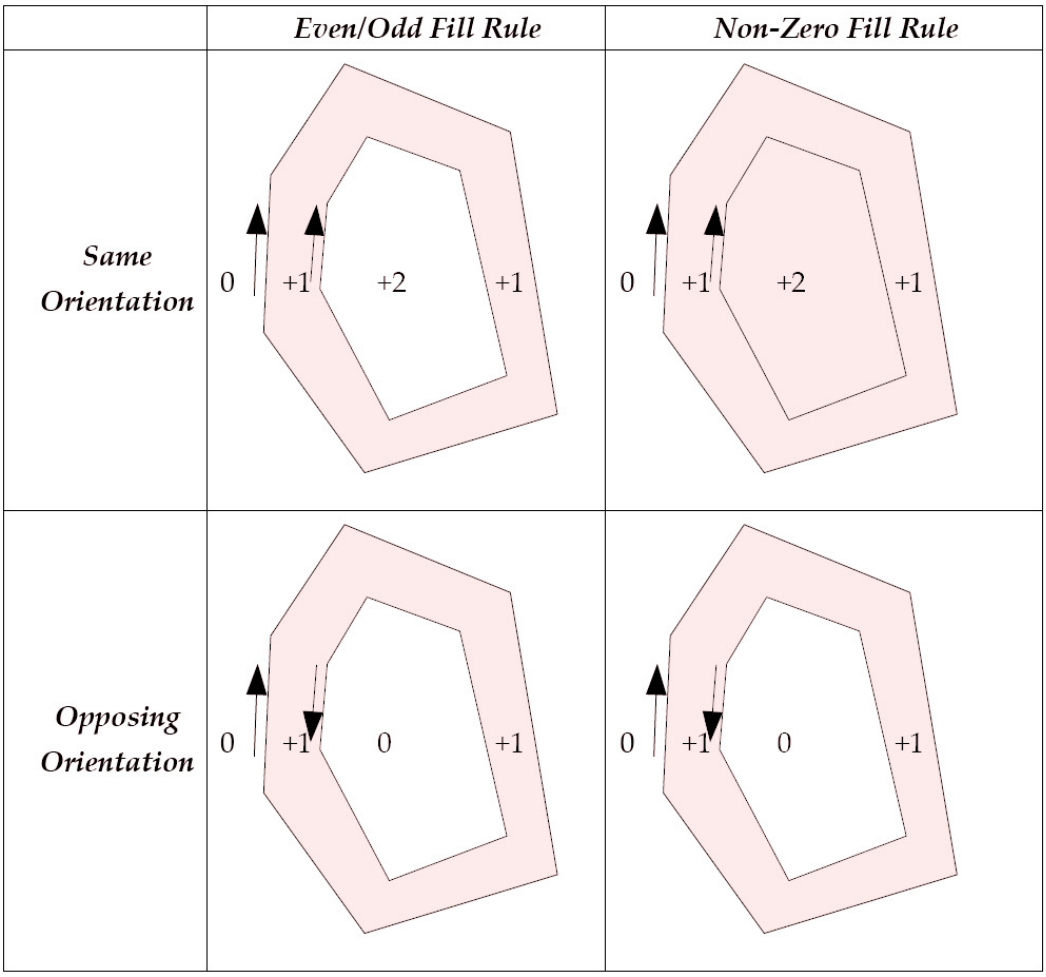


Figure 2.8: Overlapping subpaths [6].

For a simple, closed shape that does not self-intersect it is intuitively clear what is inside and what is outside. However, it is possible for OpenVG paths to self-intersect or to overlap themselves. This can

happen between different subpaths or even between parts of the one and same subpath. To be able to fill such paths consistently across implementations, it is necessary with a strict rule that defines what is the inside (should be drawn) and what is the outside (should not be drawn) of the path. OpenVG defines two such rules: Odd/even and non-zero. The application can select which one to use when rendering the path.

To decide whether a point is inside or outside, the amount of *overlap* at that point is used. The overlap is defined by the subpaths that intersect the point. Subpaths that are defined by segments in clockwise order increases the overlap, while subpaths defined in counter-clockwise order decrease the overlap. A subpath that self-intersects so that it overlaps the point multiple times increases the overlap that number of times. (This is exactly how the stencil algorithm calculates overlap - see 2.7.2)

The Odd/even fill rule states that a point with an odd overlap count is inside, otherwise it is outside. Similarly, the non-zero fill rule states that a point with non-zero overlap is inside, otherwise it is outside. Figure 2.8 shows a path that consists of two subpaths, with two different orientations and using both fill rules.

OpenVG requires implementation to perform this check correctly for paths that have up to 255 crossings along any line. Otherwise the behaviour is undefined.

2.5.4 Segment Commands

The geometry of a path is defined with an array of segment commands. Subpaths are defined by separating connected segment commands with *move to* commands. Each subpath is defined using segments of type *straight line*, *quadratic Bézier curve*, *cubic Bézier curve*, *elliptical arc* and some more that are special cases of the former.

Move To

This command starts a new subpath at the given point.

When filling paths, the previous subpath is automatically closed with a straight line back to its starting point. If a path does not start with a move to command, a move to (0,0) is assumed.

Straight Line

A straight line is drawn from the implicit starting point to the end point. (The starting point of the segment command is implicitly set by the previous segment command, so the command has only one parameter: end point.)

Quadratic Bézier Curve

Quadratic Bézier segments are defined with three points: An implicit starting point, an end point and a single control point. (The starting point of the segment command is implicitly set by the previous segment command, so the command has two parameters: Control point position and end point.)

The shape of the quadratic Bézier curve is smooth, and goes from the start point to the end point, but does not generally pass through the control point. It is always inside the control polygon defined by these three points. An affine transform to the control polygon has the same effect as transforming the curve itself.

In parametric form, the equation for a quadratic Bézier curve is:
$x(t) = x_0 * (1-t)^2 + 2 * x_1 * (1-t) * t + x_2 * t^2$
$y(t) = y_0 * (1-t)^2 + 2 * y_1 * (1-t) * t + y_2 * t^2$

where t varies from 0 to 1, (x_0, y_0) is the starting point, (x_1, y_1) is the control point and (x_2, y_2) is the end point.

The tangent at the starting point is $(x_1, y_1) - (x_0, y_0)$, and the tangent at the end point is $(x_2, y_2) - (x_1, y_1)$

Cubic Bézier Curve

Cubic Bézier segments are defined with four points: An implicit starting point, an end point and two control points. (The start point of the segment command is implicitly set by the previous segment command, so the command has three parameters: Control point positions and end point.)

The shape of the cubic Bézier curve is smooth, and goes from the starting point to the end point, but does not generally pass through the control points. The curve is always contained inside the convex hull formed by these four points. An affine transform to the control polygon has the same effect as transforming the curve itself.

In parametric form, the equation for a cubic Bézier curve is:
$x(t) = x_0 * (1-t)^3 + 3 * x_1 * (1-t)^2 * t + 3 * x_2 * (1-t) * t^2 + x_3 * t^3$
$y(t) = y_0 * (1-t)^3 + 3 * y_1 * (1-t)^2 * t + 3 * y_2 * (1-t) * t^2 + y_3 * t^3$

where t varies from 0 to 1, (x_0, x_0) is the starting point, (x_1, y_1) is the first control point, (x_2, y_2) is the second control point and (x_3, y_3) is the end point.

The tangent at the starting point is $(x_1, y_1) - (x_0, y_0)$, and the tangent at the end point is $(x_3, y_3) - (x_2, y_2)$

Elliptical Arc

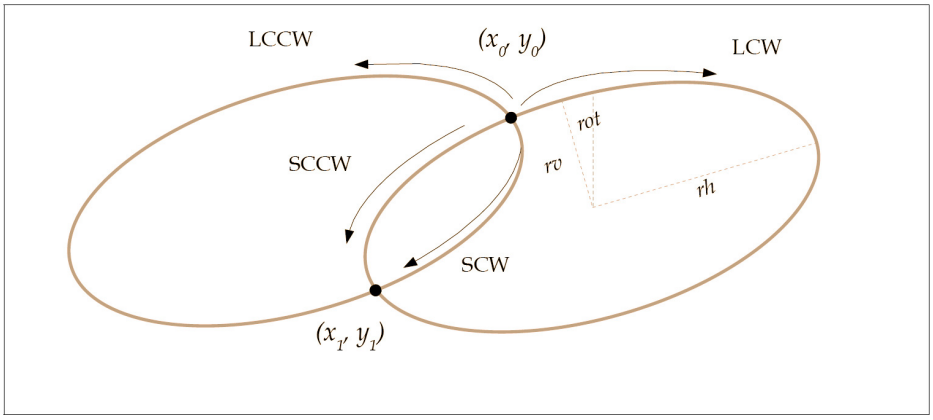


Figure 2.9: The four possible ellipse paths from starting point to end point [6].

Elliptical arcs are created by tracing a section of an ellipse from the implicit starting point to an end point. The ellipse is given by parameters for an ellipse equation: horizontal radius *rh*, vertical radius *rv* and rotation angle rot or θ. This gives four possible arcs, distinguished by their direction around the ellipse, and whether the bigger or smaller path is taken. The four possible paths around the ellipse are shown in figure 2.9.

If the there is no solution with the given parameters, the radii are scaled with the smallest uniform factor that permits a solution.

Others

Other segment commands include special cases of the former, where some of the parameters are made implicit and thus can be omitted.

2.5.5 Maximum Approximation/Rasterization Error

The OpenVG specification states that implementations are allowed to use simplified geometry when rendering, provided these rules hold:

"For purposes of estimating whether a pixel center is included within a path, implementations may make use of approximations to the exact path geometry, providing that the following constraints are met. Conceptually, draw a disc D around each pixel center with a radius of just under $\frac{1}{2}$ a pixel (in topological terms, an open disc of radius $\frac{1}{2}$) and consider its intersection with the exact path geometry:

1. If D is entirely inside the path, the coverage at the pixel center must be estimated as 1;
2. If D is entirely outside the path, the coverage at the pixel center must be estimated as 0;
3. If D lies partially inside and partially outside the path, the coverage may be estimated as either 0 or 1 subject to the additional constraints that:
   (a) The estimation is deterministic and invariant with respect to state variables apart from the current user-to-surface transformation and path coordinate geometry; and
   (b) For two disjoint paths that share a common segment, if D is partially covered by each path and completely covered by the union of the paths, the coverage must be estimated as 1 for exactly one of the paths. A segment is considered common to two paths if and only if both paths have the same path format, path data type, scale, and bias, and the segments have bit-for-bit identical segment types and coordinate values. If the segment is specified using relative coordinates, any preceding segments that may influence the segment must also have identical segment types and coordinate values."

Observe that if the simplified geometry never deviates as much as one pixel unit from the real geometry, these rules always hold.

2.6 Two Different GPU Architectures

Most of this chapter has been adapted from [35]. Chapter 2.6.3 has been added. The information is originally based on [5], [43], [45] and [19].

Modern GPUs can be divided into two types of architectures:

1. Immediate Mode Rendering
2. Deferred Rendering

While the same code can run on both types of architectures when using a standardized API such as OpenGL ES or OpenVG, they have some different characteristics related to memory traffic and performance. Performance considerations will be discussed later in chapter 4.1.3. I will now explain the difference between immediate mode and tile based GPUs.

2.6.1 Immediate Mode Rendering

This chapter is based on [18] and [4].

When an immediate mode renderer receives draw-calls from the application, it starts to execute them immediately, one at a time. The primitives are not reordered, but more than one can be processed in parallel. The primitives flow straight through the pipeline, gets converted to pixels and are written into the render target. A deferred renderer on the other hand, receives all commands needed to render a frame before it starts rasterizing.

Today, the most common renderers are immediate mode renderers. Two notable exceptions are ARM's Mali series and Imagination Technologies' PowerVR [2]. These are *tile based renderers*, which is a form of deferred renderer. Tile based renderers are explained in the next chapter.

In intensive applications, most pixels are changed more than once during the rendering of a frame, so most of the render target must be written multiple times. In the game Doom 3, each pixel is often written more than 30 times per frame [36]. In an immediate mode renderer, rendering is performed in the order which primitives are issued by the application. The writes to a specific pixel is spread across the time used to render the frame. Almost the whole render target will contain intermediate pixel values until all primitives are processed. This means that the render target pixmap can not be cached efficiently on chip.

A lot of bandwidth is wasted for transferring pixels to and from off-chip memory for this reason. Modern implementations try to reduce off-chip traffic by compressing the contents of the render target, but the amount of traffic can still be very high.

Wide data buses and memory systems with low latency are very expensive. In addition, excessive off-chip traffic generates heat and drains power. This makes immediate mode rendering especially unattractive in solutions made for handheld devices.

The advantages of immediate mode rendering are:

- Simple and well-tested approach. Brute force.
- Does not need to buffer all commands needed to render a frame.
- Can render an unlimited number of primitives with constant memory usage.

The major problem with immediate mode rendering is the high bandwidth usage due to render target traffic. This is worse in high resolution and with complex scenes with overlapping geometry.

A real-world example of an immediate mode renderer is described in the following chapter.

The GeForce 6 Architecture

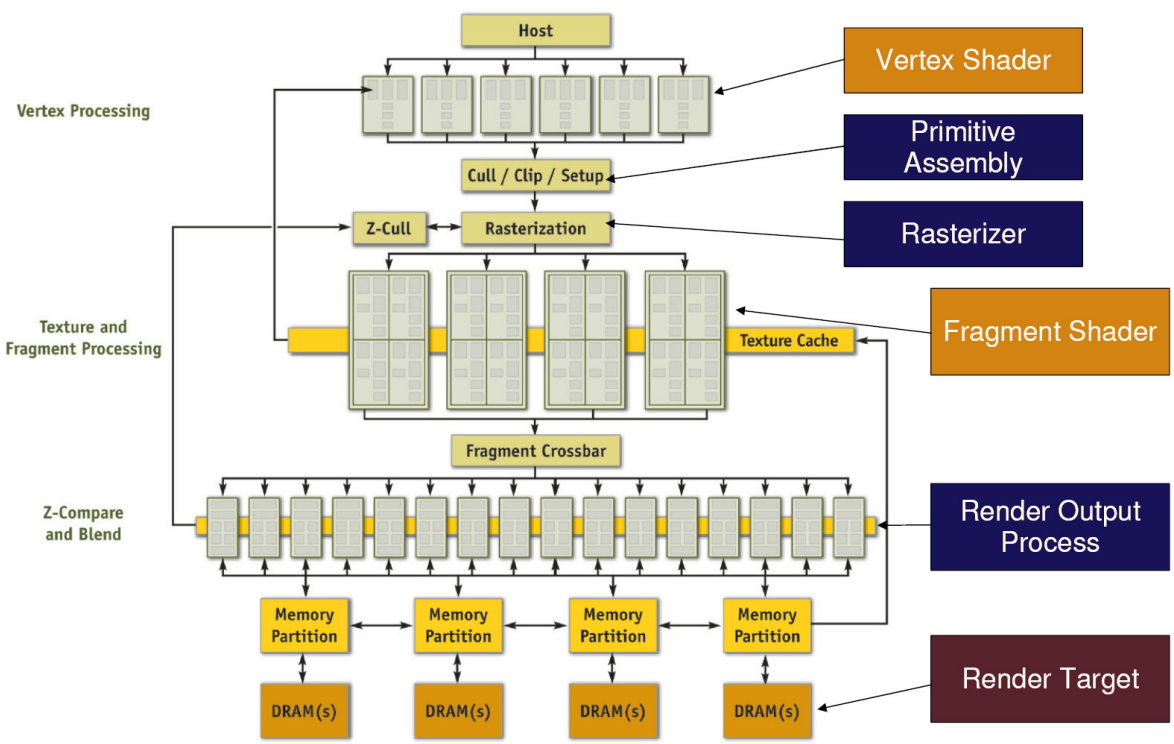


Figure 2.10: A block diagram of the GeForce 6 series architecture [31].

This chapter is based on [4].

The GeForce 6 architecture is a typical immediate mode, programmable GPU architecture. Figure 2.10 is a block diagram of this architecture. The diagram can be easily mapped to the conceptual pipeline explained in chapter 2.4.2, and illustrated in figure 2.5.

Host is the computer to which the GPU is connected. This is done through a high speed AGP or PCI Express bus. The host runs the operating system, the API and the applications using the GPU. The host sends draw-calls to the GPU, which the GPU responds to at once.

The vertices of the draw-calls are distributed among the vertex shaders. The GeForce 6 architecture has six vertex shader units.

The shaded vertices move on to the Cull/Clip/Setup unit. This unit is the same as the primitive assembly stage in figure 2.5. It combines the vertices into primitives, and removes parts of primitives that are outside the screen. The unit also sets up the primitives for the rasterizer.

The rasterization stage finds the pixels that are covered by the primitives. It can also check if a pixel is visible by checking if it is behind a pixel that was earlier drawn in the same position. This is done in cooperation with the Z-Cull unit.

The accepted pixels are distributed among the fragment shaders. The GeForce 6 architecture has 16 fragment shaders. They compute the final color of the pixel by sampling textures through the texture cache, doing light calculations, etc. This is the same as the fragment shader stage in figure 2.4.

When the pixels are fully processed by the fragment shaders they are distributed to the Z-compare and blend units through the fragment crossbar.

The Z-compare and blend units update the render target by first checking if the pixel will actually be visible. If the pixel is visible, color buffer blend will be performed. This unit can also do dithering, so it corresponds to the render output process in figure 2.5.

The updated pixel colors are in the end written back onto the memory partitions which store the render target.

The amount of data increases through the pipeline. This can be seen in the diagram by the increased parallelism needed. The architecture has only 6 vertex shaders while it needs 16 fragment shaders. This is because normally each primitive rendered needs 3 vertices, but will cover more than 3 pixels.

In the same way, the needed bus bandwidth must increase down the pipeline. The bandwidth between the host and the GPU is 8GB/s, while the GPU's memory interface has a bandwidth of 35GB/s.

2.6.2 Tile Based Rendering

This chapter is based on [18] and [19] unless otherwise noted.

A common approach to deferred rendering in hardware is *tile based rendering*. The main goal of tile based rendering is to eliminate redundant off-chip transfers of render target contents.

With tile based rendering, the render target is divided into fixed-size squares called *tiles* and then completely rendering one tile at a time. This part of the pixmap is small enough that it can be kept on chip. When it is completely rendered, the pixel values are transferred off the chip to the render target using burst writes. This reduces the memory bandwidth used for transfer of render target contents. A deferred rendering approach also enables cheap anti-aliasing since the core can keep a high-resolution version of the tile in on-chip cache, and resample it to a lower resolution when it is transferred to RAM.

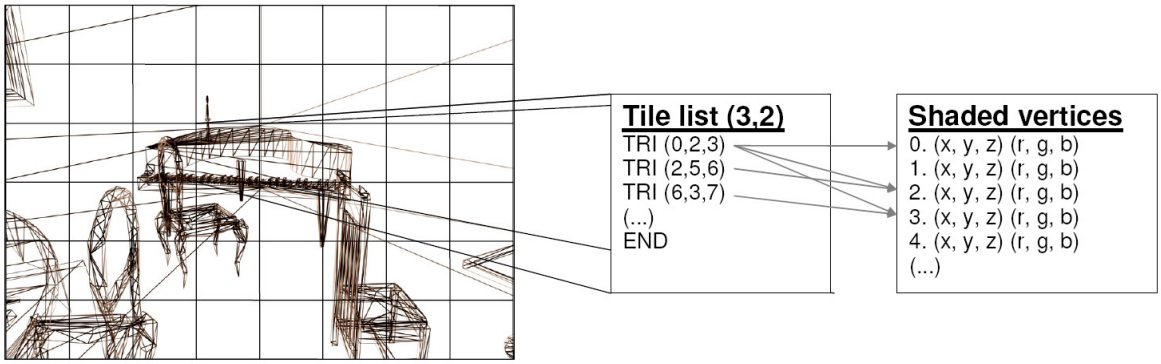


Figure 2.11: Illustration of the tile list data structures.

To be able to render one tile at a time, rendering must be delayed until all draw-calls have been received from the application. This is because each tile is rendered only once, so all primitives that intersect the tile must be known before the tile can be rendered. The result of this analysis is recorded in *tile lists*. This process is called *tiling*. It can be viewed as a sorting pass where primitives are split up and sorted according to where they are located on the screen [18]. To find the primitive's positions, the vertex shader

and primitive assembly processes must be performed. The results of the vertex shading as well as the tile lists are traditionally written to off-chip memory as it is usually too much data to be stored on chip. The tiling process is illustrated in figure 2.11.

Two forms of tiling are common: exact and bounding-box. The difference is explained in 2.6.3.

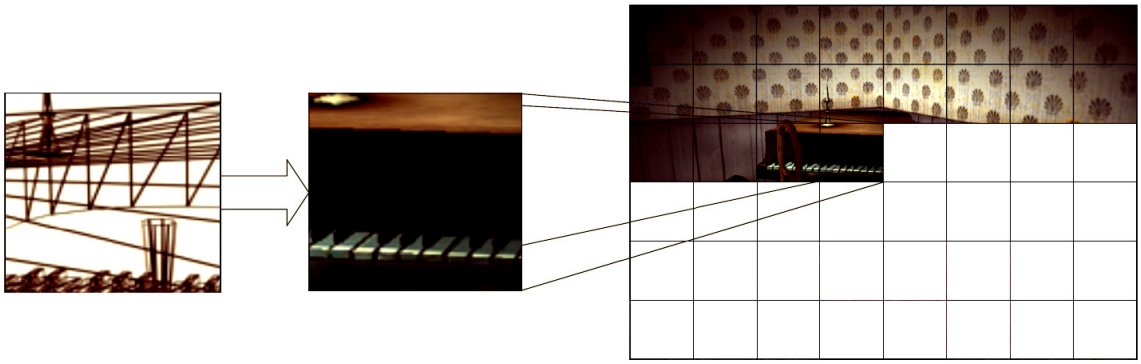


Figure 2.12: Illustration of the rendering and writeback process.

A tile based renderer consists of essentially two units: The tile list builder and the rendering unit. The draw-calls for a frame are first tiled by the tile list builder, before each tile is rendered by the rendering unit. Rendering can not begin before all draw-calls for the frame have been tiled. For each tile, the rendering is performed according to the tile lists. Each tile of the render target pixmap is stored in the core, and only written back to RAM when it is completely rendered. The transfer of the tile pixmap to the render target memory is called *writeback*. Anti-aliasing can be performed by resampling the on-chip pixmap to a lower resolution during writeback. Figure 2.12 illustrates the rendering and writeback of a single tile.

The tile list set is double buffered so that tiling of one frame can happen at the same time as another frame is rendered. This is done to be able to utilize both the tiling and the rendering hardware at the same time. The disadvantage is that the frame buffer that is displayed will be delayed one frame.

2.6.3 Exact vs. Bounding Box Tiling

This chapter is based on [19].

Tiling refers to the process of deciding which tiles can be affected by a primitive and adding tile list commands to the corresponding tile lists. This can be done *exact* - that is, the command is only added to the tiles that actually overlap the primitive. However, it can be beneficial for several reasons to apply a simpler approach. This can save die area on the chip, and can sometimes be faster. One alternative is to make a bounding box around the primitive and add the command to all tiles that intersect the bounding

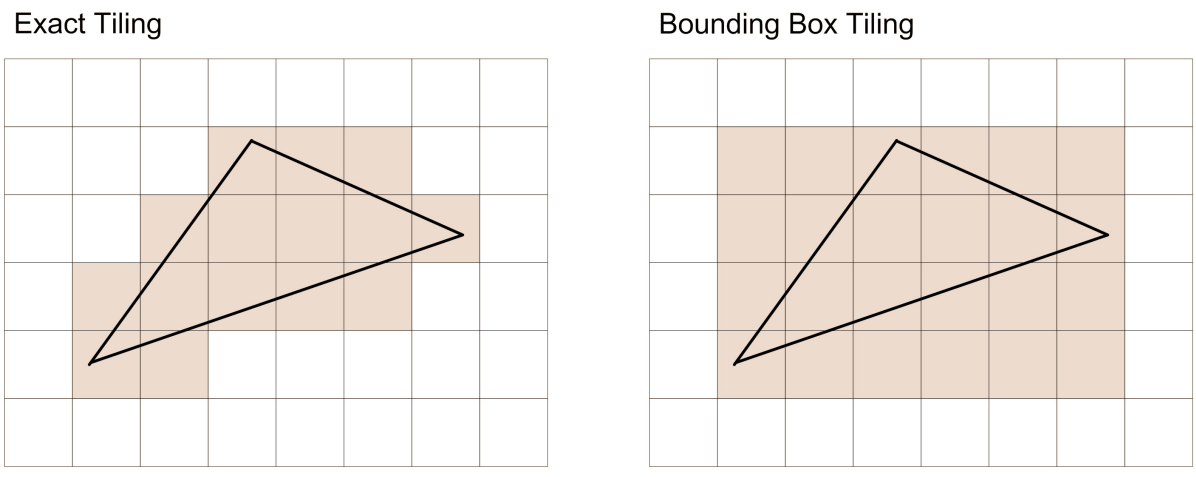


Figure 2.13: Exact tiling vs. bounding box tiling.

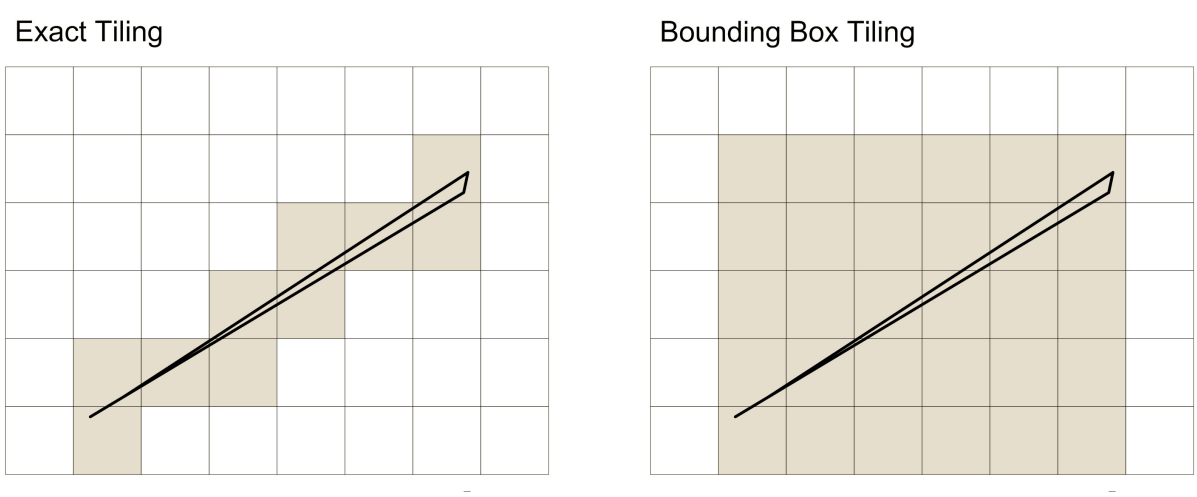


Figure 2.14: Bounding box tiling can a give bad fit.

box. For tiles that do not intersect the primitive, the command can be discarded shortly after it has been read back by the rendering device. See figure 2.13 for an example where bounding box tiling does a relatively good fit.

In most cases, bounding box tiling gives a good fit, and thus an acceptable amount of redundant tile list commands. However, some types of input geometry create a large number of redundant tile list commands. For example, thin, diagonal triangles (diagonal slivers) are suboptimal in a tile based renderer with bounding box tiling. Figure 2.14 shows a case where a lot of redundant tile list commands are created. Geometry with a large number of slivers (long, thin triangles) will not perform well on such hardware.

2.6.4 Discussion

Immediate mode renderers and tile based renderers have similar capabilities. Traditional immediate mode renderers use much memory bandwidth for frame buffer accesses, while traditional tile based renderers generate a significant amount of traffic due to shaded vertices and tile lists.

In immediate mode as well as tile based renderers, performance is affected by the number of polygons that are drawn as well as their area. For tile based renderers one may also try to restrict the number of tile-list commands since they contribute to memory traffic and usage. Chapter 4.1.3 explains how the shape of triangles affect the number of tile list commands in tile based renderers and the cacheability of the render target in immediate mode renderers.

2.7 Polygon Rasterization

A polygon can be viewed as a subpath that has only line segments. I will discuss two ways of rendering polygons on the GPU: Tessellation into non-overlapping triangles and the stencil algorithm.

2.7.1 Tessellation Into Non-Overlapping Triangles

This chapter is based on [30].

The GPU renders triangles. Rendering a polygon is just a question of dividing up it into triangles using the CPU and then rendering these with the GPU.

Self-intersecting polygons must be processed according to a fill rule to generate an equivalent set of triangles.

While tessellating simple classes of polygons (i.e. convex) can be fairly straightforward, OpenVG requires correct handling of all polygons, including self-intersections, with two different fill rules (see 2.5.3). Thus, efficient (and correct) tessellation is not a simple task and there exists a large number of

algorithms with various complexity and performance characteristics.

A tessellation of a convex polygon can be trivially created by drawing a triangle fan from the polygon's line segments towards an arbitrary point at or inside the polygon.

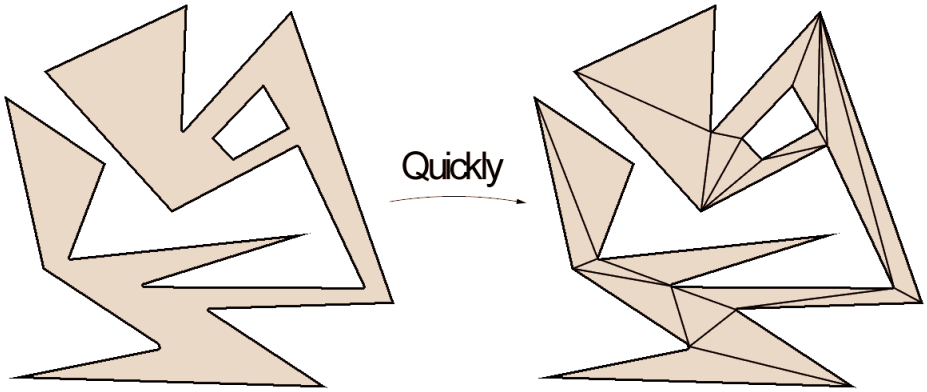


Figure 2.15: Polygon and possible tessellation.

See figure 2.15 for an example of a tessellation. I will not use tessellation algorithms for non-convex polygons in the assignment, and pseudocode is therefore not provided.

2.7.2 Stencil Algorithm

This chapter is based on 2.7.2.

The CPU overhead of concave polygon tessellation can be avoided at the cost of some potentially redundant polygon-filling in the GPU. This is accomplished by a well-known algorithm known as the stencil algorithm, described in [44]. Triangles are formed almost trivially by connecting each line segment to an arbitrary fixed pivot point, creating a triangle fan. This is equivalent to the tessellation of a convex polygon. The remainder of the algorithm is performed on the GPU using *stencil buffer* operations.

The stencil buffer is a buffer in the GPU which contains one integer for each pixel on the screen. The GPU can be configured so that when rendering a triangle, stencil buffer values covered by the triangle is either incremented or decremented. When rendering using the stencil algorithm, increment or decrement based on the *orientation* of the triangle. That is, a triangle that has its three vertices in clockwise order *increments* the stencil values, while a triangle with vertices in counter-clockwise order *decrements*.

The result is that pixels that are outside of the polygon end up with a stencil value of 0, while pixels that are inside one piece of the polygon get a stencil value of 1. Pixels that are covered multiple times by the polygon get a higher stencil value. That is, the stencil buffer contains the *overlap* at each pixel.

Finally, the polygon can be drawn into the frame buffer. OpenVG's two fill rules can be easily implemented by filling all pixels that have either odd or non-zero stencil values in the stencil buffer, respectively.

This pseudocode renders a polygon using the stencil algorithm:

1. Calculate the centroid of the polygon (A good choice for arbitrary point)
2. Disable color buffer writes.
3. Clear stencil buffer.
4. Setup stencil operations:
    - Write enabled.
    - Stencil test always passes.
    - Increment on clockwise triangles.
    - Decrement on counter-clockwise triangles.

5. for each line segment in the polygon
    - Draw a triangle using starting point, end point and centroid
6. Enable color buffer writes.
7. Setup stencil operations:
    - Write disabled.
    - Stencil test for non-zero value.
8. For non-zero fill rule, set a stencil mask of 1. For odd/even fill rule, set a stencil mask of 1. Thus, even values will appear as 0 to the stencil test, and only odd pixels will be filled.
9. Set up the GPU state for the desired paint.
10. Find the screen-space bounds of the polygon and render a quad.

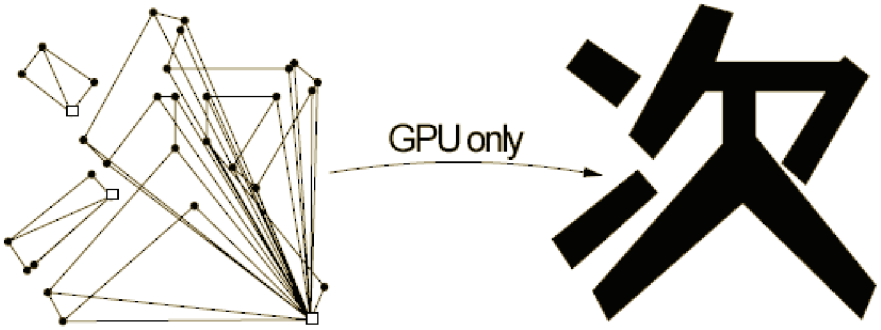


Figure 2.16: Illustration of the stencil algorithm. Based on a figure from [32].

See figure 2.16 for an example of the stencil algorithm. Note that an arbitrary vertex of the polygon is used instead of the polygon centroid as pivot for the triangle fan. This gives correct results, but usually more overdraw and thinner triangles (slivers).

The running time of the stencil algorithm (with triangle fan tessellation) is $O(n)$ with respect to the number of line segments. (Although long, thin triangles may decrease performance as described in 4.2.) However, required fill-rate increases with the complexity of the polygon. (Self-intersections, large concave sections etc.)

2.8 Recursive Subdivision of Paths

The GPU must render using triangles. A traditional approach to path-rendering is to convert the path to a polygon-approximation. A polygon is a path that has only line segments. Polygons can be rendered using triangles, as explained in the next section.

To create the polygon approximation, curved segments such as arcs and quadratic curves must be converted into to an appropriate number of short line segments along the curve. The number of lines should be high enough that there are no visible artifacts.

The OpenVG specification implies a maximum rasterization error which should be the basis for this subdivision. (See chapter 2.5.5)

Each segment can be handled individually. A simple and well-known algorithm for approximation of segments with simpler curves is *recursive subdivision*. The technique is explained in [25] and [23]. Please see listing 2.1 for an iterative implementation in C-like pseudocode.

The number of line segments generated by the algorithm is not optimal, but it is fairly good [25]. A bigger problem is that calculation of maximum error can involve quite a lot of mathematical operations for

Listing 2.1: Classic recursive subdivision algorithm (Iterative)

```
// let Segment[] segments be an input array of path segments
Segment[] segments = ..;
int i = 0;
While (i < segments.Length())
{
    // Collapse segments[i] to line using start and end point
    LineSegment lineApprox = ApproximateWithLine(segments[i]);

    // Measure maximum distance from LineApprox to segments[i]
    float maxDistance = GetMaxDistance(segments[i], lineApprox);

    // If maxDistance is larger than global threshold:
    if (maxDistance > threshold) {

        // Split the segment in two (at the middle) and assign the two
           segments to seg_l and seg_r
        (seg_l, seg_r) = Subdivide(segments[i]);


        // Overwrite segment[i] with seg_l
        segments[i] = seg_l

        // Insert seg_r at segment[i+1]
        segments.insert(i+1, seg_r);

    } else {

        // Overwrite segments[i] with lineApprox
        segments[i] = lineApprox;

        // increment iterator
        i++;
    }
}
```

some segment types, and it can therefore be relatively slow. While there are possibly faster algorithms, this is sufficient for our purposes. Algorithms for creating polygonal approximations are not easily implementable on the GPU and are usually done on the CPU.

2.9 Offset Curves

An offset curve is defined as a curve that lies at a constant *offset* pixel units away from an original curve, measured in the direction of its normal.

Offset curves are often used to generate strokes for paths. Since OpenVG supports lines, quadratic curves, cubic curves and ellipses, offset curve generation for these curve types is of high interest.

Generating the offset curve of a line is rather easy: The offset curve of a line is another line. However, the offset curve of a quadratic or cubic Bézier curve or an elliptical arc is a high-degree polynomial which generally cannot be generated or rasterized easily [47].

Elber, Lee and Kim compare various offset curve approximation methods in [22]. According to this paper, the following two approaches are suitable for low-degree Bézier curves and elliptical arcs: Tiller and Hanson [47] approximate offset curves using the same segment type as the original by offsetting the control polygon edges in the direction of their normal. J. Hoschek [28] [29] approximate offset curves using cubic Bézier segments.

Perhaps the most essential operation when applying approximation methods is the error estimation. The OpenVG specification requires that approximation and rasterization error is smaller than 1.0 in total. An error estimation function that gives the maximum distance between the approximated offset curve and an ideal offset curve is needed.

The topic of generating offset curves for stroking is large part left for future work.

2.10 Loop and Blinn's Approach for Curve Rasterization

This chapter is based on [37].

A traditional approach to path rasterization is to convert the paths to polygons and then apply a polygon rasterization technique. This is a much used approach, but is not very good for performance and memory usage.

In 2005, Charles Loop and graphics pioneer Jim Blinn introduced the idea of evaluating implicit versions of Bézier curve equations in the fragment shader. The fragment shader can then discard pixels that lie outside the curve. When rendering with their technique, only one or two triangles need to be drawn per curve segment. Quadratic and cubic Bézier curves are rendered using this technique.

Loop and Blinn use a traditional tessellation approach to correctly render interior polygons, but do not support self-intersecting paths as required by OpenVG. An alternative is to use a variant of the stencil algorithm, as suggested in [32].

2.10.1 Rasterizing Quadratic Bézier Curves

Table 2.1: Varying table for quadratic Bézier curve rendering.

Vertex	Varying Values (u, v)
Starting point	$(0.0, 0.0)$
Control point	$(0.5, 0.0)$
End point	$(1.0, 1.0)$

Form a triangle from the start, end and control points. Varyings u and v at the three vertices are set to the values in table 2.10.1 and are linearly interpolated across the triangle by the GPU. The fragment shader can now determine whether it is at the inside of the curve by evaluating the implicit equation $u^2 - v > 0$.

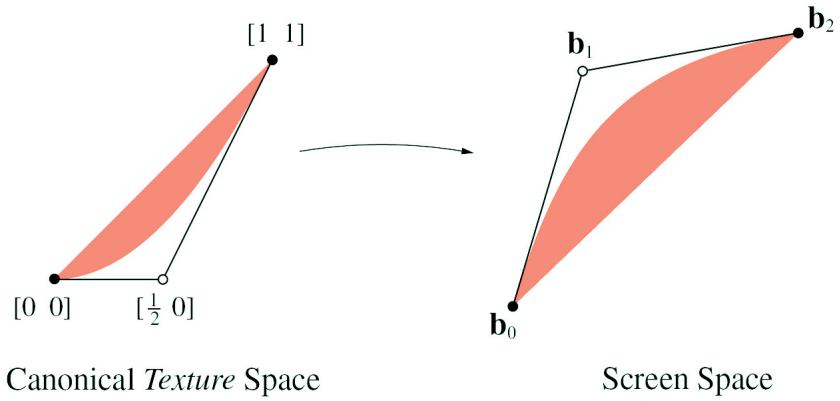


Figure 2.17: Quadratic curve equation in canonical texture space [37].

Listing 2.2: Fragment shader for rendering quadratic curves (GLSL)

```
void main()
{
    float a = gl_TexCoord[0].x*gl_TexCoord[0].x; // u^2
    float b = gl_TexCoord[0].y; // v
    if (a>b) discard; // discard pixel if u^2 > v
}
```

The mathematical background for the implicitization and the varying values will not be explained here, but an equivalent method will be developed in chapter 5.2.2. Refer to [37] for a brief explanation of the mathematics behind this technique.

Figure 2.17 shows how the quadratic curve looks in canonical texture space (u and v are the axes), and then when transformed to screen space for rendering.

A GLSL fragment shader that discards pixels that are outside the curve is given in listing 2.2. It is based on the HLSL shaders in [37].

2.10.2 Rasterizing Cubic Bézier Curves

Rasterizing cubic Bézier curves is much more complicated than quadratic Bézier curves. The underlying mathematics will not be explained here, but I give a walkthrough of the necessary steps. The systematization of the algorithm into steps is my own. Please refer to [37] for further explanation.

Step 1: Convert Bézier Control Points to Power Basis

Multiply the matrix $\mathbf{B}$ containing the Bézier control points

$$\mathbf{B} = \begin{pmatrix} x_0 & y_0 & 1 & 0 \\ x_1 & y_1 & 1 & 0 \\ x_2 & y_2 & 1 & 0 \\ x_3 & y_3 & 1 & 0 \end{pmatrix}$$

with matrix $\mathbf{M}_3$, the change of basis matrix

$$\mathbf{M}_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix}$$

thus, the power basis coefficients of the Bézier curve are:

$$\mathbf{C} = \mathbf{M}_3 * \mathbf{B}$$

Step 2: Compute the Vector d

From matrix $\mathbf{C}$

$$\mathbf{C} = \begin{pmatrix} x_0 & y_0 & w_0 & - \\ x_1 & y_1 & w_1 & - \\ x_2 & y_2 & w_2 & - \\ x_3 & y_3 & w_3 & - \end{pmatrix}$$

Define vector $\mathbf{d} = [d_0\ d_1\ d_2\ d_3]$ with

$$d_0 = \begin{vmatrix} x_3 & y_3 & w_3 \\ x_2 & y_2 & w_2 \\ x_1 & y_1 & w_1 \end{vmatrix} \quad d_1 = -\begin{vmatrix} x_3 & y_3 & w_3 \\ x_2 & y_2 & w_2 \\ x_0 & y_0 & w_0 \end{vmatrix}$$

$$d_2 = \begin{vmatrix} x_3 & y_3 & w_3 \\ x_1 & y_1 & w_1 \\ x_0 & y_0 & w_0 \end{vmatrix} \quad d_3 = -\begin{vmatrix} x_2 & y_2 & w_2 \\ x_1 & y_1 & w_1 \\ x_0 & y_0 & w_0 \end{vmatrix}$$

Step 3: Curve Categorization

The varyings at the vertices are not the same of for every cubic curve, but are calculated as a function of the control point coordinates. A step in calculating these varyings is to find a matrix $\mathbf{F}$. Different methods must be used to create this matrix depending on which of 5 categories the curve belongs to. The *test condition* given for each category determines whether a curve belongs in that category.

The orientation of the curve must also be determined in this step. Initially, clockwise orientation is assumed.

Category 1: The Serpentine
Test Condition:

$$\begin{aligned} d_1 &\ != 0, \\ 3*d_2^2 - 4*d_3*d_1 &> 0. \end{aligned}$$

Let

$$\begin{aligned} (t_l, s_l) &= (d_2 + \frac{1}{\sqrt{(3)}}\sqrt{3d_2^2 - 4d_1d_3}, 2d_1) \\ (t_m, s_m) &= (d_2 - \frac{1}{\sqrt{(3)}}\sqrt{3d_2^2 - 4d_1d_3}, 2d_1) \\ (t_n, s_n) &= (1,0) \end{aligned}$$

then,

$$\mathbf{F} = \begin{pmatrix} t_l t_m & t_l^3 & t_m^3 & 1 \\ -s_m t_l - s_l t_m & -3s_l t_l^2 & -3s_m t_m^2 & 0 \\ s_l s_m & 3s_l^2 t_l & 3s_m^2 t_m & 0 \\ 0 & -s_l^3 & -s_m^3 & 0 \end{pmatrix}$$

Vectors (t_l, s_l) and (t_m, s_m) should be scaled to unit length to avoid overflows.

If $d_1 < 0$, flip the orientation.

Category 2: The Loop,
Test Condition:

$$\begin{aligned} d_1 &\ != 0, \\ 3*d_2^2 - 4*d_3*d_1 &< 0. \end{aligned}$$

Let

$$\begin{aligned}(t_d, s_d) &= (d_2 + \sqrt{4d_1d_3 - 3d_2^2}, 2d_1)\\ (t_e, s_e) &= (d_2 - \sqrt{4d_1d_3 - 3d_2^2}, 2d_1)\end{aligned}$$

then,

$$\mathbf{F} = \begin{pmatrix} t_d t_e & t_d^2 t_e & t_d t_e^2 & 1 \\ -s_e t_d - s_d t_e & -s_e t_d^2 - 2s_d t_e t_d & -s_d t_e^2 - 2s_e t_d t_e & 0 \\ s_d s_e & t_e s_d^2 + 2s_e t_d s_d & t_d s_e^2 + 2s_d t_e s_e & 0 \\ 0 & -s_d^2 s_e & -s_d s_e^2 & 0 \end{pmatrix}$$

Problems arise when $0 < t_d/s_d < 1$ or $0 < t_e/s_e < 1$. A part of the curve that lies outside of $0 < t < 1$ intersect the visible part of the curve, making the orientation of the curve ambiguous. The segment should then be subdivided into a new pair by splitting at the offending parameter value (t_d/s_d or t_e/s_e, respectively). The sub-curves will have unambiguous orientation except at the very limits of their parameter range ($t = 0$ or $t = 1$) and can be rendered normally.

To determine orientation, calculate the following:

$$\begin{aligned}h_0 &= d_1d_3 - d2^2\\ h_1 &= d_1d_3 + d_1d_2 - d_1^2 - d2^2\\ H(\cdot) &= h_0 \; if \; |h_0| > |h_1|,\\ &\quad\; else \; h_1\end{aligned}$$

if $d1 * H(\cdot)$ is positive, flip the orientation.

Category 3a: Cusp With Inflection at Infinity,
Test Condition:

$$\begin{aligned}d_1 &!= 0,\\ 3 * d_2^2 - 4 * d_3 * d_1 &= 0.\end{aligned}$$

Boundary case between above two categories. Can be merged with category 1.

Category 3b: Cusp With Cusp at Infinity,
Test Condition:

$$\begin{aligned}d_1 &= 0,\\ d_2 &!= 0,\end{aligned}$$

Let

$$\begin{aligned}(t_l, s_l) &= (d_3, 3d_2)\\ (t_m, s_m) &= (1, 0)\\ (t_n, s_n) &= (1, 0)\end{aligned}$$

then,

$$\mathbf{F} = \begin{pmatrix} t_l & t_l^3 & 1 & 1 \\ -s_l & -3s_l t_l^2 & 0 & 0 \\ 0 & 3s_l^2 t_l & 0 & 0 \\ 0 & -s_l^3 & 0 & 0 \end{pmatrix}$$

Orientation never needs to be flipped

Category 4: The Curve is Really a Quadratic,
Test Condition:

$$\begin{aligned} d_1 &= 0, \\ d_2 &= 0, \\ d_3 &!= 0, \end{aligned}$$

Must abort and instead use the previously described approach for rendering quadratic curves.

Category 5: The Curve is Really a Line or Point,
Test Condition:

$$\begin{aligned} d_1 &= 0, \\ d_2 &= 0, \\ d_3 &= 0, \end{aligned}$$

Must abort and instead render line or point directly.

Step 4: Calculate Varyings

Multiply the matrix $\mathbf{F}$ with matrix $\mathbf{M}_3^{-1}$, the change of basis matrix

$$\mathbf{M}_3^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{1}{3} & 0 & 0 \\ 1 & \frac{2}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

producing the matrix

$$\mathbf{P} = \mathbf{M}_3^{-1} * \mathbf{F}$$

The curve can now be rendered by drawing a quad using the vertices and varyings from table 2.10.2.

Table 2.2: Varying table for cubic curve rendering

Vertex	Varying Values (k,l,m)	
-	Orientation Normal	Orientation Flipped
Starting point	$(\mathbf{P}_{0,0}, \mathbf{P}_{0,1}, \mathbf{P}_{0,2})$	$(-\mathbf{P}_{0,0}, -\mathbf{P}_{0,1}, \mathbf{P}_{0,2})$
Control point 0	$(\mathbf{P}_{1,0}, \mathbf{P}_{1,1}, \mathbf{P}_{1,2})$	$(-\mathbf{P}_{1,0}, -\mathbf{P}_{1,1}, \mathbf{P}_{1,2})$
Control point 1	$(\mathbf{P}_{2,0}, \mathbf{P}_{2,1}, \mathbf{P}_{2,2})$	$(-\mathbf{P}_{2,0}, -\mathbf{P}_{2,1}, \mathbf{P}_{2,2})$
End point	$(\mathbf{P}_{3,0}, \mathbf{P}_{3,1}, \mathbf{P}_{3,2})$	$(-\mathbf{P}_{3,0}, -\mathbf{P}_{3,1}, \mathbf{P}_{3,2})$

Step 4: Render the Boundary Polygon

Using a fragment shader which evaluates the implicit equation $u^3 < vw$, the boundary polygon is rasterized with the GPU. A convex polygon is rendered using two triangles or a quad, with the varying values specified in the previous step. A convex control polygon is guaranteed to contain the whole curve and can be easily rendered as two triangles. It is not specified in the paper how to handle a concave control polygon.

A GLSL fragment shader that discards pixels that are outside the curve is given in listing 2.3. It is based on the HLSL shaders in [37].

Listing 2.3: Fragment shader for rendering cubic curves (GLSL)

```
void main()
{
    float a = gl_TexCoord[0].x * gl_TexCoord[0].x * gl_TexCoord[0].x; // u
        ^3
    float b = gl_TexCoord[0].y * gl_TexCoord[0].z; // v*w
    if ( a>b ) discard; // discard pixel if u^3 > v*w
}
```

2.10.3 Rendering a Path

After showing how to render curve segments by evaluating an implicit equation in the fragment shader, Loop and Blinn show how this technique can be extended to render text using TrueType fonts. These are represented in a way similar to OpenVG paths, but are based only on quadratic curves and do not have self-intersections. A triangle is created for each segment, and the remaining interiors of the path is drawn with a polygon. Some limitations are present. Overlapping segments are handled in a fashion that is not guaranteed to terminate if boundary curves intersect or osculate. Tessellation is performed in CPU and can be expensive.

2.10.4 Kokojima et al's Approach

Loop and Blinn's method is enhanced in the sketch [32] mainly with a simpler and more robust polygon rasterization method. It is essentially a variant of the stencil algorithm described in chapter 2.7.2.

This removes the CPU overhead of tessellation included in Loop and Blinn's original paper. Also, termination is now guaranteed even if boundary curves intersect or osculate.

Kokojima et al claim that their method is more than 10 times faster than the approach used by Loop and Blinn for deformable paths. (This implies that caching of tessellation results is not possible.)

3 State of the Art

I have found evidence of only one implementation of Bézier curve rendering that is similar to Loop and Blinn's approach. Stefan Gustavson renders cubic Bézier curves in RenderMan by evaluating the implicit equation in an SL shader [46]. General path rendering is not a topic and special cases such as described in chapter 2.10.2 are not handled. An anti-aliasing technique is however implemented.

Kokojima et al's sketch [32] simplifies the approach and improves performance by using the stencil buffer. Apart from this sketch, I have not found any papers with references to Loop and Blinn's paper or their algorithm that are relevant for the purposes of this assignment. I have also searched for relevant keywords such as *Bézier curve*, *implicit equation* and *pixel (or fragment) shader*.

I used all the following search engines in my search for relevant material:

- google.com
- scholar.google.com
- www.engineeringvillage.com
- portal.isiknowledge.com
- citeseer.ist.psu.edu
- ieeexplore.ieee.org
- springerlink.com

Existing vector graphics renderers have been investigated to see whether some of these use interesting approaches and for comparison. Renderers that run purely in software and do not use specific hardware acceleration were not considered. There are several implementations that use OpenGL for acceleration of simpler tasks like image composition and rasterization of triangles and polygons. However, they all use the traditional approach of polygonal approximation. There is one dedicated hardware implementation, but little is known about the algorithms it uses.

The assignment text states that polygonal approximation and tessellation is a common approach to path rasterization. This chapter shows that the statement is correct.

3.1 Hardware-Accelerated Renderers

This section discusses hardware-accelerated vector graphics applications. The interesting aspect with regard to my work is the algorithms used for rasterization of paths.

3.1.1 Cairo (Vector Graphics Library)

This chapter is based on information from [17] and [1].

Cairo is an open-source graphics library which supports operations similar to PostScript or PDF. Among other things, there is support for stroking and filling paths with cubic Bézier curves.

Through a backend called the *glitz* library, Cairo gains support for hardware-accelerated rendering through OpenGL. However, glitz only helps with simple tasks such as trapezoid rasterization and image composition. Paths are still rasterized by polygonal approximation and tessellation in the CPU by the Cairo library.

3.1.2 AmanithVG (OpenVG)

This chapter is based on information from [13].

AmanithVG is an OpenVG implementation completely built on top of the OpenGL and OpenGL ES 1.x and 2.0 APIs.

Polygonal approximation and tessellation is used. The website claims that the polygonal approximation algorithm produces statistically 35% fewer segments than a classic, recursive approach. They also claim that the tessellator always produces the minimum number of triangles, but there are no claims about scalability or shape of triangles. Since no claims are made that this is accelerated by the GPU, it can be assumed that polygonal approximation and tessellation is as usual done 100% by the CPU.

3.1.3 Qt (Vector Graphics Library)

This chapter is based on information from [16] and [10].

The Qt library by TrollTech has a vector graphics component.

In a blog at [10], a person who appears to be a developer at TrollTech's graphics division reveals details about the implementation. It is apparent that polygonal approximation is being used. Unlike the other renderers reviewed here, a technique involving the stencil buffer is used for rasterizing polygons. Claims are made that this is a much faster approach than that used by competing software. One can assume that the algorithm used is, or is at least similar to, the stencil algorithm explained in chapter 2.7.2.

My technical supervisor Thomas Austad has been in personal contact with the developer who wrote the forum post. According to his e-mails, the QT rasterizer uses the stencil algorithm. His benchmarks apply animation to the paths so that results of tessellation cannot be cached across frames by the competing libraries. Renderers that use expensive tessellation techniques may end up with less overdraw, but they lose against the stencil algorithm due to the CPU overhead of tessellation [20].

3.1.4 AMD/Bitboys G12 (OpenVG Hardware Accelerator)

G12 is a hardware accelerator for vector graphics with support for SVG Tiny and OpenVG. It was originally developed and announced by Bitboys. Bitboys was later bought by ATI, which was in turn acquired by AMD. Little is known about the algorithms it uses since it is a commercial product and is not yet widely available.

3.2 Other Renderers

The following programs include vector graphics rendering. However, they use CPU-based software rasterization and thus a completely different class of algorithms than GPU-based renderers. They are therefore not discussed.

- Adobe SVG, an SVG viewer
- Adobe Acrobat Reader, a PDF viewer
- Adobe Flash, a drawing application

- Adobe Illustrator, a drawing application
- GDI+, a vector graphics library by Microsoft
- FreeType, an open-source font rendering library

4 Evaluation of Algorithms for Path (and Polygon) Rasterization

This chapter aims to find the most promising approach to path rasterization for our purposes.

The approaches presented in the background chapter will be evaluated and compared. The focus is on efficiency.

Chapter 4.1 discusses what criterions should be applied when evaluating the efficiency of an approach.

All of the path rasterization techniques that I consider include polygon rasterization as part of the algorithm. I therefore continue by evaluating different polygon rasterization techniques in chapter 4.2.

The efficiency of various path rasterization approaches are evaluated in chapter 4.3. One main approach is finally selected for further development and implementation in the following chapters.

4.1 Criterions for Evaluation of Efficiency

This chapter reflects on what characteristics affect efficiency and which statistics should be collected so that they can be considered. This has to be done before the statistics can be collected and the various approaches compared.

The most accurate and obvious solution would be to implement the approaches on the target device and measure rendering time. This is however impractical for several reasons:

Implementing an optimized version of each approach takes a lot of time. Extensive profiling is required to ensure that time is not spent doing something that can be avoided and is not directly related to the approach, such as garbage collection, redundant state changes in the GPU or overhead from the OpenGL driver layer. A prototype is implemented in this thesis, but it has not been profiled or optimized and rendering time is therefore not representable of a final implementation.

Target devices are handheld units with fixed-function and programmable GPUs. These units are just arriving or have not yet arrived on the end-user market. They are expensive and can be cumbersome to write native programs for. Numbers gathered from a desktop computer will not be representable for the target devices.

I will instead compare platform- and implementation-independent statistics such as polygon count and running time complexity.

Chapter 4.1.1 explains load balancing between GPU and CPU and tries to make a comparison on how much is available of each resource (CPU time and fill-rate) in a typical setup. Chapter 4.1.2 discusses bandwidth issues and defines some statistics that can be used to compare various algorithms' bandwidth requirements. Finally, chapter 4.1.3 discusses how the shape of triangles affect performance and concludes that long, thin triangles should be avoided.

4.1.1 Balancing CPU vs. GPU usage

The target device has two units that run in parallel: The CPU and the GPU. To maximize efficiency, load should be evenly distributed across these units. Tile based renderers have two units within the GPU itself, running in parallel: The tile-list builder and the rasterizer/renderer itself. Thus, tile list count and amount of overdraw should be such that these jobs take about the same amount of time.

Since input data and the specifications of the target device may vary from case to case, it is of course impossible to balance these loads perfectly. It should be ensured however that all these units are put to good use. GPUs typically have a lot of fill-rate, and it should be used. CPU time is on the other hand often required by the application itself, and the OpenVG implementation should therefore use as little CPU time as possible. For these reasons, the OpenVG driver should perform tasks on the GPU rather than the CPU whenever this is practical.

A typical target device has a 300mhz GPU which can draw one pixel per clock cycle. With a typical display size of 640x480, the GPU is capable of filling the whole display with pixels around 1.000 times per second. Thus, fill-rate is considered an abundant resource for our purposes, and will not even be measured.

CPU time is thus a much more limited resource than fill-rate.

4.1.2 Bandwidth Considerations

Traditional polygonal approximation approaches generate a large amount of vertices and triangle indices. Since polygonal approximation and tessellation is performed in the CPU. This data must therefore be uploaded from the CPU to GPU memory every time a path is animated or changes scale. It must be read from memory by the GPU every time it is rendered. This is a major bottleneck when using the traditional polygonal approximation method [40] [20]. Triangle count and vertex count are therefore important statistics that directly affect bandwidth usage.

Tile based renderers consist of two units working in parallel: The tile list builder and the rendering unit, as explained in chapter 2.6.2. Tile lists are written to memory by the tile list builder and then later read back by the rendering unit. This causes memory traffic. As will be explained in 4.1.3, a high number of tile list commands typically indicates slivers which is also undesirable in an immediate mode renderer.

The number of tile list commands is therefore an important statistic affecting bandwidth for tile based renderers and also to a degree immediate mode renderers.

4.1.3 About the Shape of Triangles (Slivers)

Triangles with one short and two long edges are called slivers. They are more common in 2d vector graphics than 3d graphics for the following reasons:

Three dimensional objects are designed to be viewed from all directions, and must be more or less uniformly tessellated so that the silhouette always looks smooth. In contrast, two-dimensional objects are only viewed from the front. The interiors do not require any additional geometry, while the silhouette needs high detail. Unless special measures are taken to avoid it, tessellation of polygons with many short line segments (such as resulting from polygonal approximation of smoothly curved shapes) typically introduce triangles that stretch from a short line segment at the silhouette to a point far away.

The GPU often needs to calculate the derivative with respect to x and y of values in the fragment shader. This is among other things used for filtering of texture samples in programmable and some fixed-function GPUs. These derivatives are calculated by synchronously executing the fragment shader for 4 neighbouring pixels and taking the differences between the coordinates specified as texture sample locations [45].

Some dummy pixel-shader threads must often be launched for pixels that are just outside the triangle so that derivatives can be calculated. The total length of the edges of a sliver triangle is very high compared to its area. A lot of dummy threads are therefore launched, wasting many fragment shader cycles.

Additional performance problems are different for immediate mode and tile based renderers, and will be explained separately.

Consequences of Slivers for Immediate Mode Renderers

Information about immediate mode renderer architectures is based on the description of the GeForce 6 architecture in [31].

The render target is generally very large and is therefore not stored on the same chip as the GPU itself, but is kept on dedicated RAM chips. Modifications to the render target are often read-modify-write operations, involving expensive off-chip traffic. Also, accesses to off-chip memory are best done in bursts.

Because of this, GPUs try to cache the part of the render target that is currently being modified on chip. These caching strategies may assume that subsequent accesses to the render target occur close to each other in screen space. This is generally false for slivers, and they therefore produce a large amount of off-chip memory traffic.

In addition, immediate mode renderers such as the GeForce 6 compress frame buffer contents to save memory bandwidth [31]. Compression and decompression takes extra time and works on square blocks of pixels. This means that whole blocks must be loded, processed and stored even though only a few pixels are to be modified.

Slivers are thus undesirable in immediate mode renderers.

Consequences of Slivers for Tile Based Renderers

In tile based as well as immediate mode renderers, rendering time is affected by both the number of triangles and pixels to be drawn. In tile based renderers, the number tile list commands is also very important.

Chapter 2.6.2 describes how tile list commands are written to memory in the tiling stage and read back during rendering. The number of tile list commands therefore affects both memory usage and the amount of memory traffic.

A sliver can intersect a large number of tiles even though its area is low. A large number of tile-list commands must be written to memory and read back. This is bad for memory usage and traffic, and affects performance.

In renderers with bounding box tiling, diagonal slivers create an excessive number of tile list commands, and is thus especially undesirable. See figure 2.14 for an example of this.

4.2 Tessellation Into Non-Overlapping Triangles vs. The Stencil Algorithm

Since all the path rasterization methods I will consider require polygon rasterization at some point, I will now evaluate two popular approaches to rendering polygons on the GPU: The stencil algorithm and tessellation into non-overlapping triangles. The algorithms themselves are described and explained in the background chapter 2.7.

Algorithms for tessellation of possibly self-intersecting polygons into non-overlapping triangles are rather expensive in terms of CPU usage, but uses the minimum possible amount of fill-rate. The stencil algorithm is however extremely cheap CPU-wise, but uses at least the double amount of fill-rate in the GPU:

- It is a two-pass algorithm, and every pixel is therefore touched at least two times: One time for finding overlap and one time for drawing in the color buffer.
- Redundant operations may be performed on pixels while finding overlap, such as incrementing and then decrementing again.

This leads me to conclude that tessellation into non-overlapping triangles is the most efficient approach if a polygon is to be rasterized multiple times. The output of the tessellation algorithm can then be calculated once and used for rasterizing the polygon any number of times, avoiding redundant overdraw. However, the stencil algorithm is probably more efficient if the polygon is to be rasterized only once, since the expensive CPU processing is avoided. I make these conclusions based on the assumption that CPU time is a much more limited resource than fill-rate. (See chapter 4.1.1.)

4.2.1 Avoiding Slivers

The most trivial tessellation algorithms as well as the triangle fan approach used in most descriptions of the stencil algorithm create a large amount of slivers. Delaunay tessellations are optimal with respect to triangle shape without inserting new points. Some tessellation algorithms, such as the ones described in [30] and [34], avoid slivers by adding Steiner points in the geometry.

Slivers are bad for performance, as explained in chapter 4.1.3.

The common variants of the stencil algorithm triangulate the polygon using a single triangle fan, which generates slivers. (All descriptions of the stencil algorithm that I have found use this approach.) It is possible to modify the stencil algorithm so that it creates triangles with a more beneficial shape by using a less trivial triangulation method. Note that normal tessellation algorithms that are meant for convex polygons can be used for triangulation with the stencil algorithm. A linear time algorithm is desirable since our goal is to rasterize complex paths as quickly as possible. An efficient linear-time algorithm that produce few slivers is presented in chapter 5.5.

4.3 Evaluation of Path Rasterization Algorithms

I described in the Background chapter several ways of rasterizing paths:

- Rasterization with the CPU (not discussed)
- Polygonal approximation and tessellation
- Polygonal approximation and the stencil algorithm
- Loop and Blinn's approach, using a Delaunay tessellation algorithm for interior polygons.
- Kokojima et al's approach (Loop and Blinn's approach combined with the stencil algorithm instead of Delaunay tessellation)

In this chapter, I will try to decide which approach is best suitable for the purpose of this assignment. In addition to the techniques described in the Background chapter, I will introduce and evaluate the idea of implementing support for curved primitives in the rasterizer.

4.3.1 Polygonal Approximation

The traditional approach to path-rendering on the GPU is to create a polygonal approximation, as explained in chapter 2.8, and then render the polygon using either the stencil algorithm or some kind of tessellation in the CPU.

The OpenVG specification requires rasterization to be correct down to one pixel unit (see chapter 2.5.5). If approximated geometry is used to render the path, the silhouette must never deviate more than this distance from the actual path. Polygon approximations of curved segments thus typically need a high amount of short line segments.

When such a polygon is to be rendered, at least one triangle will be created for each line segment. This approach therefore leads to a very large number of vertices and triangles, and slivers are hard (or expensive) to avoid.

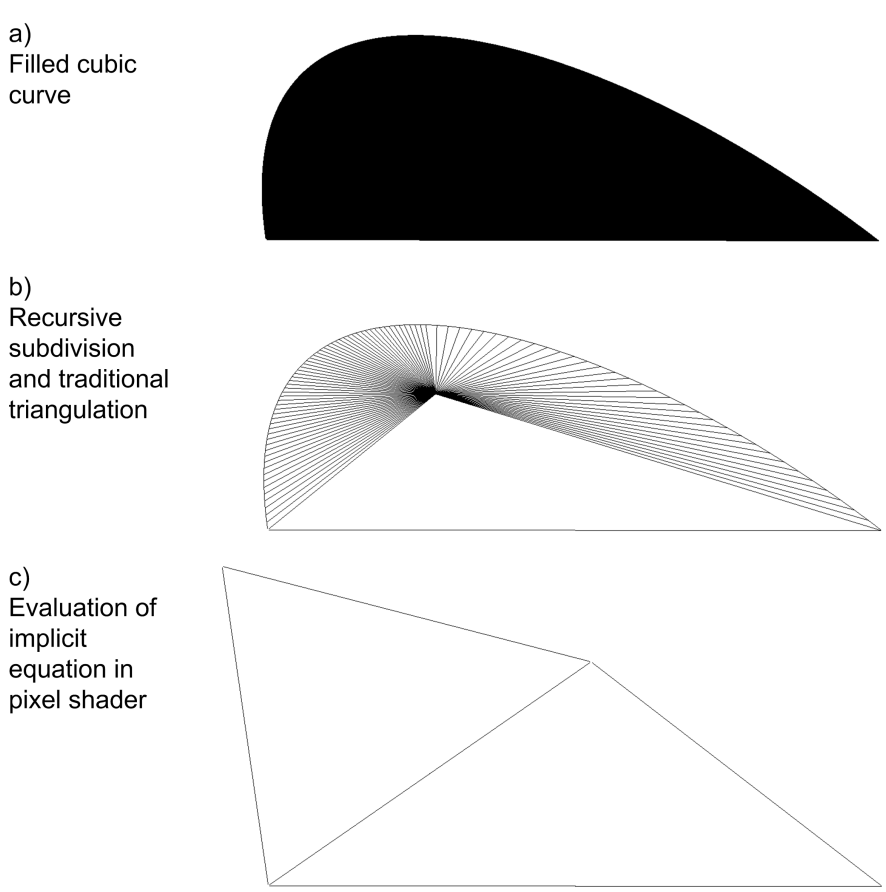


Figure 4.1: Wireframe view of cubic curve, rendered with two different approaches.

This approach was used by all the existing vector graphics software packages discussed in chapter 3.1.

4.3.2 Loop and Blinn's Approach with Delaunay Tessellation

This chapter discusses the approach presented in [37], described in chapter 2.10.

The new technique by Charles Loop and graphics pioneer Jim Blinn [37] demonstrates how programmable GPUs can be used to render not only line segments, but also quadratic and cubic segments using only simple geometry. They draw a coarse bounding polygon around each segment and use a fragment shader to discard pixels that are outside the curve. This can reduce vertex and triangle count dramatically compared to the polygon approximation approach described above. Also, CPU overhead can be lower because subdivision can be avoided.

Figure 4.1 shows a cubic curve rendered with two different techniques. Image a) shows the curve as it should appear after rendering. Image b) shows a wireframe view of the curve as if rendered using polygonal approximation. Finally, image c) shows that it can be rendered with only two triangles using Loop and Blinn's approach.

Loop and Blinn do not allow overlapping triangles to be generated in their approach. To overcome this problem, they subdivide offending segments until they do not overlap. This approach does not guarantee termination, and the search for overlapping triangles is non-trivial.

For rendering interior polygons, they use a constrained Delaunay tessellation. The details of their approach is not explained, and the operation is performed in the CPU with a significant overhead. (Kokojima et al claim that their approach is more than 10 times faster when rendering a path only once [32].) A method such as [30] or [34] can probably be used to avoid slivers in the triangulated result. The algorithm does not support self-intersecting polygons, such as required by OpenVG.

4.3.3 Kokojima et al's Approach

This chapter discusses the approach presented in the sketch [32] and described in chapter 2.10.4.

Kokojima et al take Loop and Blinn's approach and apply what is basically an adapted version of the stencil algorithm.

The approach promises efficient and robust path rendering without the CPU overhead and problems associated with tessellation of possibly self-intersecting polygons. It also avoids the special case for concave curve segments in the original approach as well as the robustness issue described at the end of chapter 4.3.2.

Kokojima et al's sketch does not describe in detail how curve rendering and interior polygon generation is done to create the correct stencil buffer output. I will describe a functioning approach in detail in chapter 5.4.

As explained above in chapter 4.2, the stencil algorithm has a tendency to produce slivers that have performance issues. These concerns still apply, but to a less extent than with the polygon approximation since the interior polygon has much fewer line segments than the result of a polygonal approximation. Fewer and more well-shaped triangles therefore result from the stencil algorithm.

OpenVG requires support for rendering complicated geometry with two fill rules. This is easily achieved with the stencil algorithm as explained in chapter 2.7.2. At least an eight bits stencil buffer is required to support OpenVG's requirement to support 255 crossings over any line in the path.

4.3.4 Conclusions

Loop and Blinn's approach for curve rasterization (with or without Kokojima et al's modifications) appears to have a clear advantage over traditional polygonal approximation techniques both in terms of vertex count, triangle count and CPU overhead.

Kokojima et al claim that their approach using the stencil buffer is more than 10 times faster than Loop and Blinn's original approach for deformable paths [32].

Delaunay tessellation as used by Loop and Blinn's algorithm involves significant CPU overhead. Kokojima et al's version is therefore much faster despite the stencil algorithm's redundant overdraw when a path is drawn only once. However, if the same path is drawn more than one time, the original approach has the possibility to store tessellation results and use them multiple times. If the same path is rasterized a sufficient number of times, the overhead from redundant overdraw in Kokojima et al's approach will no longer be neglible, and the original approach will be faster.

This is analogous to the discussion of tessellation vs. the stencil algorithm in chapter 4.2. My conclusion is also similar: Loop and Blinn's original approach is most efficient if a path is to be rasterized a large number of times, while Kokojima et al's variant is beneficial for rasterizing a path only one or a few times.

If this problem is solved, an advanced OpenVG implementation may switch between the two techniques based on how many times a path is expected to be drawn.

I will focus on using only Kokojima et al's variant. The main reasons for choosing this approach over Loop and Blinn's original approach are:

- Loop and Blinn's original approach does not in its current form support self-intersecting polygons such as required by OpenVG.
- I assume that CPU time is a much more valued resource than fill-rate. (See chapter 4.1.1.)
- It promises high performance in all common cases.
- It is easier to implement.

5 Novel Approaches and Improvements to Algorithms

The previous chapter concluded that Kokojima et al's variant of Loop and Blinn's approach from 2005 was the most promising of the considered approaches for efficient path rasterizing.

However, as we shall see in chapter 5.1, some limitations remain before it can be used for rendering paths according to the OpenVG specification.

Additions and improvements to the approach will be presented in chapter 5.2, among other things removing the mentioned limitations.

The idea of adding hardware support for Loop and Blinn's approach to fixed-function GPUs is presented in chapter 5.3.

A thorough description of how path rasterization can be performed using Kokojima et al's approach is given in chapter 5.4 since their own sketch is not very detailed.

Finally, a new tessellation algorithm that produces fewer slivers than the traditional approach is presented in chapter 5.5.

5.1 Critical Issues of the New Approaches When Applied to OpenVG

Although Loop and Blinn's approach and Kokojima et al's variant of the same is very efficient, some issues remain until it can be used for a conformant OpenVG implementation. I will now describe the problems that I have identified. They are solved in later subchapters.

Support for Both Fill Rules
Kokojima et al's paper describes a version of the stencil algorithm which implements an odd/even fill rule. OpenVG also needs support for non-zero fill rule. This will be solved in chapter 5.4.

Support for All Segment Types
Loop and Blinn only describe techniques for rendering quadratic and cubic curves. In addition to these segment types, OpenVG also supports elliptical arcs. Although they could surely be approximated well using Bézier curves, by following Loop and Blinn's line of thought, it is fairly simple to create a fragment shader based approach for rendering elliptical arcs and other segment types. The method will be explained in chapter 5.2.1.

Support for Fixed-Function Hardware
The requirement specification presented in chapter specifies that the algorithms be implementable on fixed-function GPUs. The required techniques are presented in chapter 5.2.3.

Constrained Rasterization Error

The OpenVG specification has clear requirements to error bounds in the rasterization.

Loop and Blinn's paper (see [37]) claims that their approach to curve rasterization is resolution indepen-

dent. However, this is the case only if one assumes that the hardware operates with unlimited precision. In reality, the method will produce severe artifacts when large segments are rasterized at sufficiently high resolution.

While high-end desktop GPUs today typically have 24 or 32 bits floating point representations, handheld GPUs may have as little precision as one part in 1024 for floating point numbers [12], and artifacts are therefore very common and apparent. This may explain how Loop and Blinn could ignore these issues.

A method for calculating and constraining rasterization error is presented in chapter 5.2.5.

5.2 Extensions and Additions to Loop and Blinn's Approach

In chapter 4.3.4, I concluded that Kokojima et al's variant of Loop and Blinn's approach was the most suitable for the purposes of this assignment. However, some issues need to be resolved before this approach can be used in a conformant OpenVG implementation.

I will extend the method to be able to completely support elliptical arcs as required by OpenVG in chapter 5.2.1. Loop and Blinn's method for curve rasterizing requires a programmable GPU. In chapter 5.2.2, I will develop a slightly different way of rasterizing quadratic curves that works better than Loop and Blinn's version on a platform with limited precision. In chapter 5.2.3 I present a new variant that works on all OpenGL ES 1.1-compatible hardware. Finally, the difficult issue of guaranteed rasterization error bounds is solved in chapter 5.2.5.

5.2.1 Rasterization of Elliptical Arcs

Any ellipse can be created by applying scale, rotation and translation to the unit circle. Matrix M is specified in [6], and represents the transform of the unit circle into an ellipse according to OpenVG's elliptical arc representation.

$$\mathbf{M} = \begin{pmatrix} rh\cos\theta & -rv\sin\theta & cx \\ rh\sin\theta & rv\cos\theta & cy \\ 0 & 0 & 1 \end{pmatrix}$$

The inverse is

$$\mathbf{M}^{-1} = \begin{pmatrix} \frac{\cos\theta}{rh} & \frac{\sin\theta}{rh} & -\frac{cx\cos\theta+cy\sin\theta}{rh} \\ -\frac{\sin\theta}{rv} & \frac{\cos\theta}{rv} & \frac{cx\sin\theta-cy\cos\theta}{rv} \\ 0 & 0 & 1 \end{pmatrix}$$

M^{-1} can be used to transform coordinates into unit space where an implicit equation (for the unit circle) can be efficiently evaluated. Since M^{-1} represents an affine transformation, the resulting unit space coordinates vary linearly in screen space. This means that they can be calculated at vertices of the control polygon and then interpolated linearly by the GPU. The fragment shader then only needs to evaluate the implicit equation of the unit circle.

The unit circle is represented by the implicit equation $\sqrt{u^2+v^2} = 1$. Squaring both sides and seeing that the equation's left side is less than one for points that are inside the circle, we have the condition $u^2+v^2 < 1$ which tests whether a point is inside the unit circle. A GLSL fragment shader that evaluates this equation and discards pixels that are not inside the curve is given in listing 5.1.

5.2.2 Improved Precision for Quadratic Curve Rendering

Loop and Blinn show how the parametric equation for quadratic curves can be implicitisized, producing an implicit equation and the varying values that are used as vertex inputs to their rasterization approach. The implicit equation and varying values are listed in chapter 2.10.1 and visualized in figure 2.17.

I found that an implicitization I performed on my own gave the same implicit equation but different varying values. While Loop and Blinn's values are all in the positive quadrant, my values are centered

Listing 5.1: Fragment shader for rendering elliptical arcs (GLSL)

```
void main()
{
    float a = gl_TexCoord[0].x*gl_TexCoord[0].x; // u^2
    float b = gl_TexCoord[0].y*gl_TexCoord[0].y; // v^2
    if ( a+b > 1.0 ) discard; // discard pixel if u^2+v^2 > 1
}
```

around $(0,0)$, are symmetric around the y-axis and lie on axes. Symmetry around the y-axis is beneficial since it enables the sign-bit to be used in the internal floating point representation, effectively doubling precision on the x-axis. Also, when rendering with the fixed-function technique (see chapter 5.2.3), this allows using mirrored textures to double the precision. The values are also easier to analyze with respect to precision since they lie on axes.

Implicitization of Quadratic Equation

In parametric form, the equation for a quadratic Bézier curve is:
$x(t) = x_0 * (1-t)^2 + 2 * x_1 * (1-t) * t + x_2 * t^2$
$y(t) = y_0 * (1-t)^2 + 2 * y_1 * (1-t) * t + y_2 * t^2$

Consider a quadratic curve with start and end points at $(-1,1)$ and $(1,1)$, and the control point at $(0,-1)$. Inserting these coordinates into the equation and solving for y, we get
$y(x) = x^2$

Observe that in the case of these control point coordinates, y(x) has only one solution for any x. Thus, $x^2 - y = 0$ is an implicitization of this special case, which is the same result that Loop and Blinn got. The condition $x^2 > y$ is true when a point is inside the curve.

Performing an affine transformation to a Bézier curve's control points is equivalent to transforming the curve itself. Thus, these values can be used as varyings and thus represent any quadratic curve with the same algorithm and fragment shader as before. (See table 5.2.2.)

Figure 5.1: Quadratic curve equation in canonical texture space.

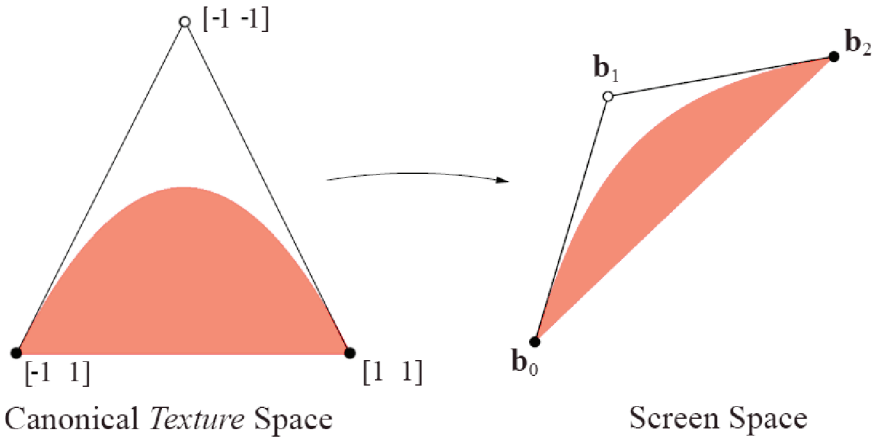


Table 5.1: Varying table for quadratic Bézier curve rendering with improved precision

Vertex	Varying Values (u,v)
Starting point	$(-1.0, 1.0)$
Control point	$(0.0, -1.0)$
End point	$(1.0, 1.0)$

Figure 5.1 shows how canonical texture space is mapped to screen space when rendering a quadratic curve with my implicitization. Compare with figure 2.17.

5.2.3 Curve Rasterization on Fixed-Function Hardware

Quadratic curves and elliptical arcs can be evaluated on fixed-function hardware by using a texture as a look-up table for the implicit equation. The equation for cubic curves takes three parameters that vary wildly in range. While a 3d texture could be used, it is unpractical since it would take up too much space to get usable precision.

The textures can be generated on demand and do not have to be stored on the device, but they will occupy memory while the OpenVG implementation is in use.

The size to use for the texture is a compromise between precision and memory usage. It is best to use a 1-bit per pixel texture format if the device supports it. If the device only supports texture formats with multiple bits per pixel, the pixels should contain a coverage value. This is bilinear filtered by the texture mapper when it is almost out of precision and creates an approximation of the over-sampled data. If a low bit-per-pixel texture format is not available, it is also possible to use a compressed texture format such as ETC (Ericsson Texture Compression). Only a subset of the ETC standard is required to store a lossless 4-bit grayscale image, so the texture can easily and quickly be generated directly in compressed format when needed. (See the specification for ETC at [15].)

Figure 5.2: Elliptical arc look-up texture.

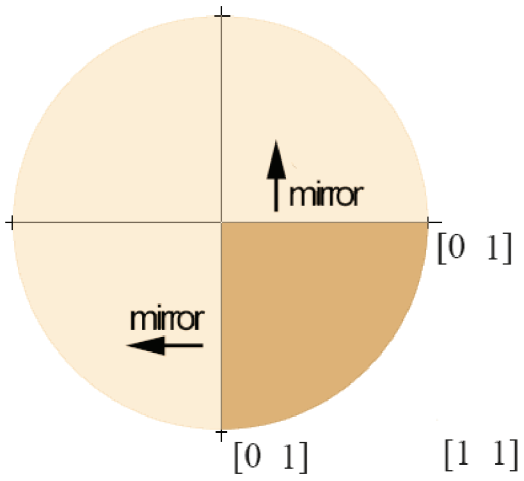
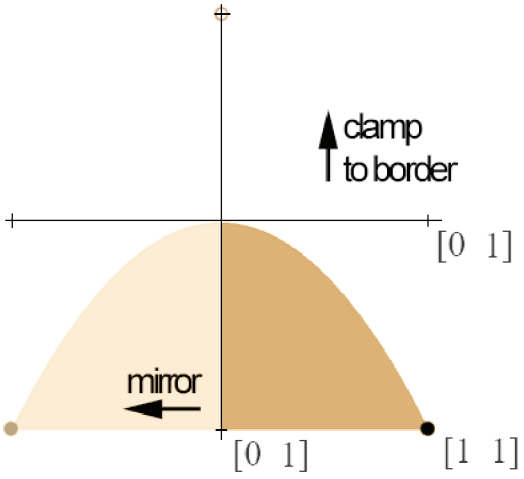


Figure 5.3: Quadratic Bézier curve look-up texture.



Figures 5.2 and 5.3 shows look-up function textures for ellipses and quadratic curves. Note that both mirroring and clamping is used to reduce the size of the texture data to $\frac{1}{4}$.

When rasterizing using this technique, the varying values are sent in as texture coordinates. Bilinear filtering can be used and will improve quality, especially if the texture has multiple bits per pixel and contains coverage values. Mip-mapping should be turned off. Alpha testing with a threshold of 0.5 is used to discard pixels that are outside the curve. Stencil testing is set up as usual.

5.2.4 Correct Rasterization of Segments With Concave Control Polygons

I did not get my implementation of Loop and Blinn's approach to correctly rasterize cubic curves with concave control polygons. The problem is described in chapter 7.3.2.

A preliminary solution to this is simply to detect the case and then subdivide. Since my path rasterizer is built around recursive subdivision and the case is easy to detect, this is a simple fix. I subdivide offending segments until they are either convex or are small enough that they can be approximated by a quadratic curve within the allowed error threshold.

It may be possible to solve this analytically and thus avoid extra subdivision in these cases. This task is left for future work.

5.2.5 Consideration of Rasterization Error

Segments are rasterized by drawing simple boundary polygons when using Loop and Blinn's approach. The CPU calculates both vertex positions and varyings for the polygon, and the GPU's rasterizer linearly interpolates these. For each pixel, a small program called a fragment shader is launched, and is given the interpolated varyings as inputs. It evaluates an implicit function and kills pixels that are outside the desired segment.

An OpenVG implementation must consider rasterization error due to limited GPU precision, as explained in chapter 5.1.

If cases where the precision becomes insufficient can be detected, they can be subdivided to reduce the error. The resulting segments will be smaller and require less precision to render correctly. (Note: For elliptical arcs, a Bézier curve must be used for approximation, as precision does not improve when subdividing.)

The OpenVG specification requires that the distance between the rasterized and the original shape is always less than maxDistance. It is therefore necessary to express the error in pixel units. Given this error plus additional approximation errors, offending segments can be subdivided and/or approximated with other segment types to keep the total error below maxDistance. A pessimistic estimate is sufficient for our purposes.

About the maxDistance Constant

The OpenVG specification includes requirements that limit the maximum approximation and rasterization error. I observed in chapter 2.5.5 that the requirements are satisfied if the rasterized shape never deviates one or more pixel units from the real geometry. I will use the name maxDistance this error threshold.

OpenVG has two modes of rendering: FASTEST and BEST, chosen by the application. For the FASTEST setting, the fastest possible rasterization that is still according to the specification should be performed. maxDistance should therefore be set to 1.0, as required by OpenVG. This allows for rough approximations while never reaching or exceeding 1 pixel error. However, with a setting this high, smooth curves do not look smooth, and the result is not visually pleasing as is discussed in chapter 10.2.9.

A lower value for maxDistance should therefore be used when rendering with the BEST setting. While the rasterization is correct according to the specification with a value of maxDistance$=1.0$, the visual quality of the output is now of concern. The value should be as high as possible, but low enough that no unpleasing artifacts are visible. Thus, the value should be selected by visual inspection. This is discussed in chapter 10.2.9.

Assumptions

For elliptical arc and cubic curve rasterization, varyings must be calculated in floating point by the CPU. Since CPUs have significantly higher precision than GPUs in most systems, error from calculation of varyings is not included in my estimation.

I will make two assumptions about the GPU's handling of floating point numbers.

- None or neglible error is introduced by the GPU's interpolation of varyings.
- All arithmetic results are rounded to the closest floating point representation.

The first point is reasonable because interpolation is usually done with high precision and in a way that introduces little error. It also implies that lines are rasterized with zero error.

Error in Floating Point Calculations

The fragment shader expressions consist of multiplications, additions/subtractions and comparisons.

Arithmetic operations such as multiplication and addition/subtraction introduce rounding error. Assuming that the GPU always rounds results to the closest floating point representation, the relative error introduced by rounding is in the worst case $r = 2^{-mantissaBits-1}$.

The relative error introduced by multiplication is in the worst case given by $a+b+r$, where a and b are the relative errors of the operands, and r is the rounding error.

The absolute error introduced by addition/subtraction is in the worst case given by $a+b+r$, where a and b are the absolute errors of the operands, and r is the absolute error from rounding.

Comparison has the same characteristics as addition/subtraction, but introduce no rounding error.

Although I assume that no error is introduced by the interpolation of varyings, inputs to the fragment shader are still rounded to the nearest floating point representation. Rounding error must therefore be applied to all the fragment shader inputs.

Precision in the Programmable GPU Rasterizer

Loop and Blinn's curve rasterization technique for programmable GPUs works by interpolating a set of varyings over the boundary polygon and using the fragment shader to determine whether each pixel is inside or outside the curve. The fragment shader performs some arithmetic operations to determine this.

I want to limit rasterization error to below maxDistance. To do this, any two samples taken with a distance of maxDistance must produce unique results for all such intermediate values. (Exception: If an intermediate value is supposed to be constant along the line formed by the two sampling points, the results should of course be equal, not unique.)

The maximum rasterization error for a segment is found by examining the interpolation of each varying along several edges. The precision of floating point numbers vary depending on how close the number is to 0. I am not interested in having more precision at some point in the curve and less precision somewhere else. To be able to reason as if using fixed point numbers, I find the maximum value of the exponent and assume that this is used for the whole edge.

The error caused by the initial rounding of interpolated varying values is found using a function CountUniqueValues. This function takes the value of the varying at the start and end of an edge, and returns the number of (evenly spaced) unique floating point representations between these values. The maximum rasterization error is found by dividing the length of the edge by this number.

A special case occurs when the varying value is equal at the start and end of the edge. The value should then be constant along the edge, and the rasterization error can thus be set to 0. (If this special case is not accounted for, the algorithm will return infinite error)

Arithmetic operations on floating point numbers in the fragment shader lead to the result being rounded to the nearest floating point representation. The error is increased by 50% because of this. Multiplication of floating point numbers increase relative error only from rounding of the result. The result of comparison is either true or false and normally does not introduce error. Although the fragment shader for drawing elliptical arcs has an addition, I will not need to take this into account other than the rounding error.

I then look at the fragment shader and count the number of arithmetic operations applied to each varying before the comparison. The rasterization error should be increased by 50% for each operation to account for rounding.

The rasterization error along an edge is found by the GetErrorAlongEdge function:

```
1
2  // number of mantissa bits in the GPU's internal floating point
       representation.
3  const int mantissaBits = 10; // 10 for FP16
4
5  // Count unique (evenly spaced) floating point values between a and b.
6  float CountUniqueValues(float a, float b)
7  {
8      // Get maximum exponent
9      int a_exp = floor(log2(fabs(a)));
10     int b_exp = floor(log2(fabs(b)));
11     int max_exp = max(a_exp, b_exp);
12
13     // Handle the special case of 0-arguments (log2 breaks down)
14     if (a==0) max_exp = b_exp;
15     if (b==0) max_exp = a_exp;
16     assert(a || b);
17
18     // Normalize with same value so that both numbers are between 0 and 2^
           mantissaBits.
19     float scale = pow(2.f, mantissaBits + (1 - max_exp));
20
21     // Number of unique, evenly spaced values
22     return fabs(a - b) * scale;
23 }
24
25 // Find the rasterization error caused by a specific varying along an edge
26 float GetErrorAlongEdge(float a, float b, float edgeLength, int
       numberOfRoundings)
27 {
28     // special case when they are equal
29     if (a==b) return 0;
30
31     // max rasterization error from interpolation
32     float maxRasterizationError = edgeLength / CountUniqueValues(a, b);
33
34     // take roundings in the fragment shader into account (increase error
           by 50% for each)
35     maxRasterizationError *= pow(1.5, numberOfRoundings);
36
37     // return the result
38     return maxRasterizationError;
39 }
```

I will now give pseudocode for GetMaxRasterizationError for quadratic, cubic and elliptical arc segments.

The quadratic case has constant varying values, so most of the calculations can be omitted. For the sake of clarity, I have not done this. I measure the rasterization error along the edges of the control polygon and choose the highest value. Since the vertices of the control polygon represent extreme values, this gives the correct maximum rasterization error.

```
1  // Returns maximum rasterization error along an edge, examining
       interpolation of u and v and accounting for rounding in the fragment
       shader
```

```
float Quadratic::GetMaxErrorAlongEdge(Vector2 a, Vector2 b, float
    edgeLength)
{
    // Reminder the fragment shader calculates: u*u > v

    // 1. Find errors for u. (u goes through one rounding)
    float u_error = GetErrorAlongEdge(a.x, b.x, edgeLength, 1);

    // 2. Find errors for v. (v goes through zero roundings)
    float v_error = GetErrorAlongEdge(a.y, b.y, edgeLength, 0);

    // done
    return max(u_error, v_error);
}

float Quadratic::GetMaxRasterizationError()
{
    // The varyings are constant for quadratics:
    Vector2 t0(-1.0, 1.0); // Starting Point
    Vector2 t1( 0.0,-1.0); // Control Point
    Vector2 t2( 1.0, 1.0); // End Point

    // edge sp<->cp
    float e0_error = GetMaxErrorAlongEdge(t0, t1, (cp-sp).GetMagnitude());

    // edge cp<->ep
    float e1_error = GetMaxErrorAlongEdge(t1, t2, (cp-ep).GetMagnitude());

    // edge sp<->ep
    float e2_error = GetMaxErrorAlongEdge(t0, t2, (ep-sp).GetMagnitude());

    // done
    return max(e0_error, e1_error, e2_error);
}
```

The cubic case is assumed to have a convex control polygon. (See chapter 7.3.2 for the reason.) As for quadratic curves, I measure the rasterization error along the edges of the control polygon and choose the highest value. Since the vertices of the control polygon represent extreme values, this gives the correct maximum rasterization error.

```
// Returns maximum rasterization error along an edge, examining
    interpolation of u, v and w and accounting for rounding in the fragment
    shader
float Cubic::GetMaxErrorAlongEdge(Vector3 a, Vector3 b, float edgeLength)
{
    // Reminder the fragment shader calculates: u*u*u > v*w

    // 1. Find errors for u. (u goes through two roundings)
    float u_error = GetErrorAlongEdge(a.x, b.x, edgeLength, 2);

    // 2. Find errors for v. (v goes through one rounding)
    float v_error = GetErrorAlongEdge(a.y, b.y, edgeLength, 1);

    // 3. Find errors for w. (w goes through one rounding)
    float w_error = GetErrorAlongEdge(a.z, b.z, edgeLength, 1);

    // done
    return max(u_error, v_error, w_error);
}
```

```
18
19  float Cubic::GetMaxRasterizationError()
20  {
21      // The varyings for vertices sp, cp0, cp1 and ep are stored in t0, t1,
            t2 and t3 respectively.
22
23      // edge sp<->cp0
24      float e0_error = GetMaxErrorAlongEdge(t0, t1, (cp0-sp).GetMagnitude());
25
26      // edge cp0<->cp1
27      float e1_error = GetMaxErrorAlongEdge(t1, t2, (cp1-cp0).GetMagnitude())
            ;
28
29      // edge cp1<->ep
30      float e2_error = GetMaxErrorAlongEdge(t2, t3, (cp1-ep).GetMagnitude());
31
32      // edge sp<->ep
33      float e3_error = GetMaxErrorAlongEdge(t0, t3, (ep-sp).GetMagnitude());
34
35      // done
36      return max(e0_error, e1_error, e2_error, e3_error);
37  }
```

For elliptical arcs, I measure the rasterization error along the horizontal and vertical axes of the ellipse and choose the highest value. Since these coordinates represent extreme values, this gives the correct maximum rasterization error.

This gives constant varying values as in the quadratic case, and most of the calculations can be omitted. For the sake of clarity, I have not done this.

Note that the code uses the word edge, but I am now really considering axis-aligned line segments, not edges of the control polygon as in the case of quadratic and cubic curves.

```
1   * Returns maximum rasterization error along an axis, examining
        interpolation of u and v and accounting for rounding in the fragment
        shader
2   float EllipticalArc::GetMaxErrorAlongEdge(Vector3 a, Vector3 b, float
        edgeLength)
3   {
4       // 1. Find errors for u. (u goes through two roundings)
5       float u_error = GetErrorAlongEdge(a.x, b.x, edgeLength, 2);
6
7       // 2. Find errors for v. (v goes through two roundings)
8       float v_error = GetErrorAlongEdge(a.y, b.y, edgeLength, 2);
9
10      // done
11      return max(u_error, v_error);
12  }
13
14  float EllipticalArc::GetMaxRasterizationError()
15  {
16      // Varyings are constants because I measure the error along the
            horizontal and vertical axis of the ellipse.
17      // The radii along these axes are specified with the hr and vr
            parameters of the elliptical arc.
18
19      // Reminder the fragment shader calculates: u*u + v*v > 1
20      // u and v both go through two roundings: one multiplication and one
            addition
21
```

```
22      // u along horizontal axis (u goes from 0 to 1, v is constantly 0)
23      float u_error = GetErrorAlongEdge(0, 1, hr, 2);
24
25      // v along vertical axis (v goes from 0 to 1, u is constantly 0)
26      float v_error = GetErrorAlongEdge(0, 1, vr, 2);
27
28      // done
29      return max(u_error, v_error);
30  }
```

Precision in the Fixed-Function Rasterizer

Fixed-function curve rasterization works much like programmable pipeline curve rasterization, but uses a texture as a look-up table for the implicit function.

Memory usage and other hardware concerns naturally limit the dimensions of the texture. When sampling the texture, the GPU hardware multiplies the texture coordinates with the texture's dimensions and rounds them to integers. When drawing a triangle that is very large on the screen, using a sufficiently small look-up texture, the same row or column will be sampled by neighbouring pixels. This can produce the same result for two points even if they are on different sides of the curve.

In fact, the only thing that needs to be changed from the programmable pipeline case is the CountUniqueValues function. Note that the lut dimension is now included in the parameter list. This is because luts for different segment types may have different sizes, and may not be square. The methods that call this function must of course also be changed to supply the correct lut dimension in the parameter list. The lutDimension value should be the real dimension of the texture, independent of any mirroring/clamping tricks.

```
1   // Count unique (evenly spaced) texture coordinates between a and b.
2   float CountUniqueValues(float a, float b, int lutDimension)
3   {
4       // Number of unique, evenly spaced texture coordinates
5       return fabs(a - b) * lutDimension;
6   }
```

The rest of the algorithm is identical to what was described for the programmable pipeline in chapter 10.2.4.

5.3 Hardware Support For Loop and Blinn's Approach

By adding some hardware for evaluating the given implicit equations, Loop and Blinn's algorithm can be used on fixed-function hardware without large look-up textures. In particular, the rasterization of cubic curves can be made to work on fixed-function hardware.

A unit inside the rasterizer would evaluate the implicit equation in the same way as the fragment shader does in Loop and Blinn's paper. This involves multiplications and comparisons, and thus requires a non-neglible amount of extra circuitry. Since input values are between -1 and 1, (Remember to normalize when rendering cubic curves,) fixed-point arithmetic can be used, and precision can be inserted where needed so that the result is good enough that subdivision is avoided most of the time, while hardware cost is kept at a minimum.

From a software and performance perspective, this approach would work much like the programmable GPU solution. I will therefore not refer to this solution explicitly in benchmarks.

The approach by Kokojima et al is still applicable to this approach.

There are two potential benefits:

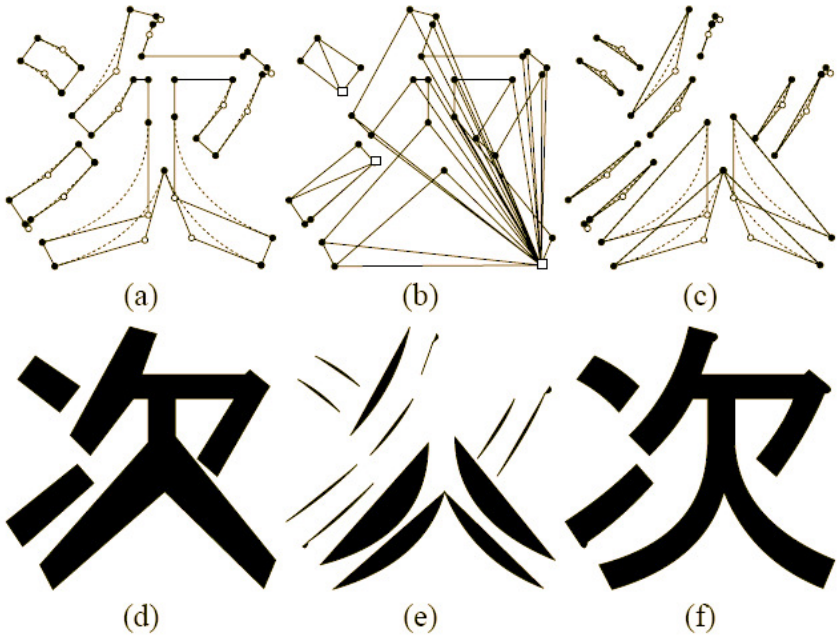


Figure 5.4: Illustrations for Kokojima et al's approach [32].

- A fragment shader is not needed. Can implement the algorithm on a modified fixed-function GPU. This saves die area, cost and power compared to upgrading to a programmable GPU.
- It may be possible to find an algorithm that can identify whole blocks of pixels that are outside the curve, and then discard them all. A fragment shader can only discard one pixel at a time.

This approach will have most of the same issues as the Loop/Kokojima approach with evaluation in the fragment shader. They can probably be solved in the same way.

Estimating the cost of this extra hardware is left for future work.

5.4 Rasterizing Paths Using Kokojima et al's Approach

Kokojima et al do not explain in detail how they use Loop and Blinn's approach to correctly rasterize complex paths. They also do not support multiple fill rules.

I will now describe an algorithm based on their sketch that correctly rasterizes complex paths with both fill rules, including intersections.

Image a) in figure 5.4 shows a path consisting of 3 subpaths with both quadratic curves and line segments. Start and end points are shown as black dots while control points are shown as white dots. Control polygons are drawn with solid lines while the curves are shown stippled. Image f) shows the desired result of the algorithm. Although this example does not include self-intersections or subpaths defined in counter-clockwise order, the algorithm will work for these cases also.

The OpenVG path is first split into its individual subpaths according to the *move to* segment commands. Since path is to be filled, all subpaths that do not start and end at the same position are closed with a line segment.

The first step is to rasterize the *interior polygon* of each subpath into the stencil buffer using the stencil algorithm. The interior polygon is constructed by drawing a line between the start and end point of each subpath segment.

Image b) in figure 5.4 shows triangles as produced by the stencil algorithm. One triangle is created for each segment towards a fixed, arbitrary point, here shown as a white square. Image d) shows the desired result of this stage. Since there are no self-intersections, and subpaths are defined in clockwise order, the black areas should have a value of 1 in the stencil buffer while the white areas should contain a 0.

The second step is to rasterize all curved segments using Loop and Blinn's approach. There is no need to handle convex and concave curve segments differently such as described in [37]. The stencil buffer is incremented when the control polygon is defined in clockwise order, and decremented when they are defined in counter-clockwise order. Concave segments thus reduce the area of the shapes produced in the first step, digging dents in the interior polygons. Convex segments increase area of the shapes, adding curved bevels around them.

Image c) shows how triangles are constructed from the control polygons of all quadratic curve segments. Image e) shows which pixels in the stencil buffer that will be modified. The vertices are in the same order as the control polygon is defined, which means that concave curves will subtract from the stencil buffer while convex ones will increment. Image f) shows the result in the stencil buffer after step 1 and 2 have been performed.

A representation of the path is now in the stencil buffer. The values correspond to the overlap at each pixel, and the path can be drawn into the color buffer using either non-zero or even/odd fill rule. This is done in the same way as in the final stage of the stencil algorithm. (See chapter 2.7.2.)

5.5 The Dividing Triangle Method for the Stencil Algorithm

The traditional version of the stencil algorithm described in [44] triangulates the polygon using a triangle fan towards a single arbitrary point. The average of the polygon's vertex positions is often used for this purpose. As mentioned in 4.2, this has a tendency to create slivers, and often diagonal ones. This is bad for tile based renderers, especially those that use bounding-box tiling.

I will now present a simple triangulation method that generates triangles of a more beneficial shape while still applying the same number of decrementing and incrementing operations to each pixel. It generates a non-overlapping tessellation for a convex polygon.

Let the polygon be stored as a list of points. The end point will be implicitly connected with the start point. If the polygon results from an open path that is to be implicitly closed, the start and end points are likely to be far from each other, while if it results from a closed path, they are likely to be as close to each other as any other neighbouring points. I will initialize the algorithm in a slightly different way depending on whether the polygon results from an open or closed path.

The dividing triangle algorithm can be easily described as a recursive algorithm, but it can also be efficiently implemented iteratively, as the only data structure that is generated is an index list which can be used directly as input to the GPU. It has linear running time, and can be performed simultaneously as writing triangle indices to GPU memory.

The algorithm works like this: Split the shape in two at the middle with a triangle. The two resulting pieces are then again split at the middle with a triangle, and so on. C-like pseudocode for the recursive version is given in listing 5.2.

Commented source code for the iterative version can be found in the file Poly.cpp of the prototype, with the method name Poly::RenderDividingTriangle.

Figure 5.5 compares the triangles generated by the dividing triangle (image b) and triangle fan triangulation (image c). Image a shows the result. Notice that the slivers are avoided. There are instead more beneficially formed triangles.

In a renderer with 16x16 tiles and bounding box tiling, this shape requires 92 tile list commands with dividing triangle triangulation, and 128 commands with triangle fan triangulation towards the centroid.

Listing 5.2: Triangulation with the dividing triangle approach (Recursive)

```
1
2  // this is the polygon, stored as a list of points
3  Vector2[] points;
4
5  void Split(int sp, int ep) {
6      // Abort if only a line remains
7      if (ep-sp < 2) return;
8      // Create the triangle
9      int pivot = (ep+sp)/2;
10     DrawTriangle( points[sp], points[pivot], points[ep % points.size()] );
11     // Recurse at the two edges of the triangle
12     Split(sp, pivot);
13     Split(pivot, ep);
14 }
15
16 // Use this function if the polygon is the result of an open path
17 void RenderDividingTriangle_Open()
18 {
19     // Initial triangle closes the gap of the open path
20     Split(0, points.size());
21 }
22
23 // Use this function if the polygon is the result of a closed path
24 void RenderDividingTriangle_Open()
25 {
26     // Split the shape at the middle
27     Split(0, points.size()/2);
28     Split(points.size()/2, points.size());
29 }
```

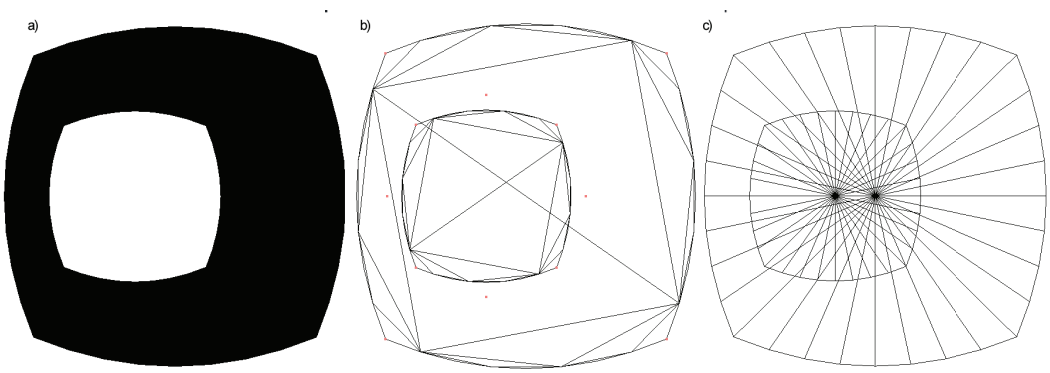


Figure 5.5: Stencil algorithm triangulation methods.

6 Path Rasterizer Architecture and Prototype Implementation

In this chapter, I put all the algorithms together and describe a complete solution for robust and efficient rasterization of paths in conformance with the OpenVG specification. The design is guided by the requirement specification from chapter , based on the algorithms that were considered most efficient in the conclusions of chapter 4, and with the improvements and additions presented in chapter 5.

Chapter 6.1 presents the main idea for the path rasterizer. An extensive description of the new, efficient OpenVG path rasterization approach is given in chapter 6.2. The implementation of the prototype itself is finally described in chapter 6.3. A short summary of what has been accomplished so far in the thesis is provided in chapter 6.4.

6.1 Introduction/Basis

The main focus of the assignment is to create an OpenVG rasterizer that uses the GPU and is more efficient than a traditional polygonal approximation implementation.

Based on the conclusions from chapter 4, my approach to efficient path rasterization will be based on Kokojima/Loop's approach described in 2.10.4, which again uses Loop and Blinn's approach for rasterization of Bézier curves by evaluating an implicit equation in the fragment shader. These algorithms were made more suitable for an efficient OpenVG implementation through a number of additions and improvements presented in chapter 5.

Loop and Blinn's approach sometimes requires segments to be subdivided due to various reasons. Recursive subdivision is traditionally used by path rasterizers for creating polygonal approximations, but my approach will be capable of rasterizing curves directly using Loop and Blinn's approach, so this stage is not required. However, I will apply the recursive subdivision algorithm to handle subdivisions due to various reasons in one consistent way. The traditional recursive algorithm for polygonal approximation is described in chapter 2.8.

I will include support for turning off and on all the novel features of my approach. In specific, it will be possible to configure the path rasterizer so that it approximates paths using only line segments. It will also be possible to switch between the dividing triangle and triangle fan triangulation methods for the stencil algorithm for filling interior polygons. Thus, the implementation can be configured to work like a traditional path rasterizer with recursive subdivision. This will ease evaluation and benchmarking of the new path rasterizer.

6.2 Description of the New, Efficient Approach to OpenVG Path Rasterization

A path consists of one or more subpaths, and each subpath is simply defined as a list of segments. When rendering a path, its subpaths are first rasterized into the stencil buffer using Kokojima et al's variant of

Loop and Blinn's approach. The path is then drawn into the stencil buffer by drawing a simple quad and using the stencil test functionality of the GPU to discard pixels that are not part of the path, according to one of the fill rules defined by OpenVG.

A segment can not always be directly rasterized for one or more of various reasons. Segments must therefore sometimes be approximated using simpler segment types that can be directly rasterized. This will be explained in chapter 6.2.2.

My procedure for efficient rasterizing of OpenVG paths using the stencil algorithm is as follows:

1. Disable color buffer writes.
2. Clear stencil buffer.
3. Setup stencil operations: Write enabled. Stencil test always passes. Increment on clockwise, decrement on counter-clockwise triangles.
4. for each subpath
   (a) Convert to only rasterizable segments. (ApproximateWithRasterizable)
   (b) For each rasterizable segment:
      i. Rasterize the segment using Loop and Blinn's approach.
   (c) Rasterize the interior polygon using the dividing triangle approach.
5. Enable color buffer writes.
6. Setup stencil operations: Write disabled. Stencil test for non-zero value.
7. For non-zero fill rule, set a stencil mask of all 1s. For odd/even fill rule, set a stencil mask of 1. Thus, even values will appear as 0 to the stencil test, and only odd pixels will be filled.
8. Set up the GPU state for the desired paint.
9. Find the screen-space bounds of the path and render a quad to fill the path.

Each segment type has its own class that inherits from the base class Segment. Conversion to rasterizable segments is done by a method ApproximateWithRasterizable, which is abstract in the Segment class and is overloaded by all the segment types. It returns a list of segments which are rasterizable, and which can be directly rasterized with a total error less than maxDistance. I will explain how ApproximateWithRasterizable works in chapter 6.2.2.

See figure 6.1 for a class diagram of the segment types.

See listing 6.1 for C++-like pseudocode implementing the path rasterization approach. The path is declared as Segment[][]. This can be read as equivalent to the C++ type std::vector< std::vector< Segment* > > - a two-dimensional dynamic array. I have used real OpenGL calls to illustrate exactly which render states are used for the GPU. In the case of programmable GPU rendering, a simple vertex shader which just passes all vertex attributes through as varyings is used.

6.2.1 Additional Optimizations

In the above explanation and the pseudocode, some compromises are done to improve clarity and readability at the expense of efficiency or generality.

Omit Redundant Start/End Points

According to the class diagram and the pseudocode, the starting point and end point of each segment is stored in attributes in the object. Since the OpenVG specification ensures that the end point of a segment is always equal to the start point of the next segment in a subpath, one of the points is redundant. It can therefore be omitted in a real implementation. One possibility is to use a linked list in such a way that a segment can access its neighbours' starting and/or end points. Another possibility is to supply the implicit point of the segment in the parameter lists when executing methods that need access to it.

Listing 6.1: Path Rasterizer (Rasterizes in white color)

```
void RasterizeFill(Segment[][] path, int fillRule) {
    // Disable color buffer writes, clear stencil buffer and setup stencil
       ops
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
    glClear(GL_STENCIL_BIT);
    glEnable(GL_STENCIL_TEST);                  // enable stencil tests
    glEnable(GL_STENCIL_TEST_TWO_SIDE_EXT); // use two-sided stencil
    glActiveStencilFaceEXT(GL_FRONT);           // modify clockwise state
    glStencilOp(GL_INCR, GL_INCR, GL_INCR); // clockwise increment
    glStencilMask(~0);                          // all bits enabled
    glStencilFunc(GL_ALWAYS, 0, ~0);            // always pass
    glActiveStencilFaceEXT(GL_BACK);            // modify counter-clockwise
    glStencilOp(GL_DECR, GL_DECR, GL_DECR); // counter-clockwise decrement
    glStencilMask(~0);                          // all bits enabled
    glStencilFunc(GL_ALWAYS, 0, ~0);            // always pass

    // For each subpath
    foreach Segment[] subpath in path {
        // Convert to only rasterizable segments
        Segment[] rasterizableSubPath;
        foreach Segment seg in path {
            Segment[] segApprox = seg.ApproximateWithRasterizable();
            rasterizablePath.Append(segApprox); // append approximation
        }

        // For each rasterizable segment
        foreach Segment seg in rasterizableSubPath {
            // Render using Loop and Blinn's approach.
            seg.Render();
        }

        // Generate interior polygon
        Vector2[] interiorPolygon
        foreach Segment seg in rasterizableSubPath {
            interiorPolygon.Insert( seg.sp );
        }

        // Render interior polygon using stencil algorithm
        RenderPolygon( interiorPolygon );
    }

    // Enable color buffer writes
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

    // Choose stencil test mask based on fill rule
    if (fillRule==ODD_EVEN) stencilTestMask = 1;          // test only LSB
      else if (fillRule==NON_ZERO) stencilTestMask = ~0; // test all bits

    // Setup stencil operations
    glDisable(GL_STENCIL_TEST_TWO_SIDE_EXT);// don't use two-sided stencil
    glStencilMask(0);                           // no stencil write
    glStencilFunc(GL_NOTEQUAL, 0, stencilTestMask); // test value != 0

    // Set up the GPU state for the desired paint.
    glColor3f(1.0, 1.0, 1.0); // White paint

    // Render a quad at the screen-space bounds of the path
    DrawQuad( GetBounds(path) );
}
```

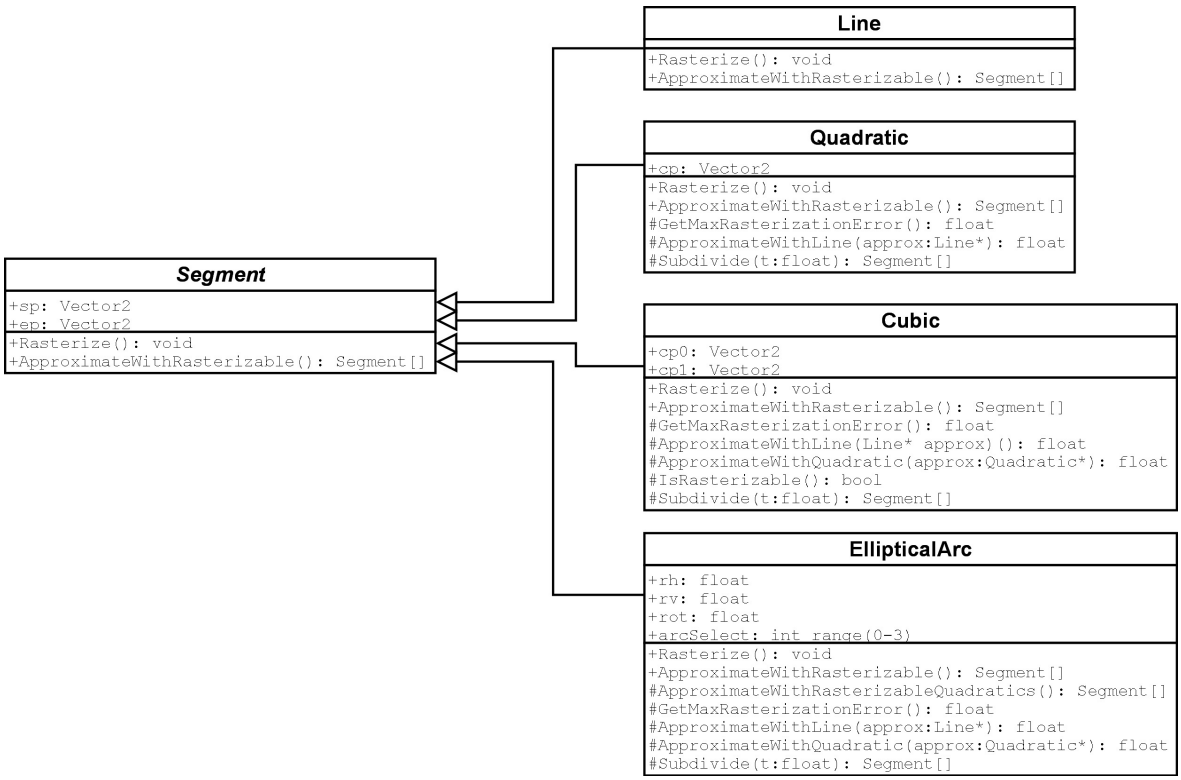


Figure 6.1: Class diagram of the segment types.

Reduce Number of GPU State Changes

In the explanation above, the prototype implementation as well as listing 6.1, segments are rasterized in the order they are defined in the path. Depending on the input, this can involve a large number of state changes, which is inefficient for some GPUs. Segments should instead be sorted by type and then rendered in one batch for each type. To keep the pseudocode and prototype clean, I have not done this here. However, it is trivial to do and should be done in a final implementation.

Remove Dependency on Two-Sided Stencil Operations

In the explanation above as well as listing 6.1, I am depending on an OpenGL extension to specify that triangles defined in clockwise order should increment the stencil buffer, while counter-clockwise triangles should decrement. If this functionality is not available in the target GPU, the same effect can be achieved in a less efficient way through the following approach:

1. Set stencil operation to increment
2. Turn on culling of counter-clockwise geometry (Triangles defined in counter-clockwise order will not be drawn)
3. Draw geometry
4. Set stencil operation to decrement
5. Turn on culling of clockwise geometry (Triangles defined in clockwise order will not be drawn)
6. Draw geometry

This is a well-known approach used to achieve different stencil operations depending on the orientation of each triangle.

6.2.2 Rasterization of Segments

By building on Loop and Blinn's approach, I have now found a way to rasterize all of OpenVG's segment types on programmable GPUs. However, they can not always be directly rasterized due to OpenVG's maximum error requirement. (See chapter 2.5.5.) Also, I have not found any feasible way to directly rasterize Cubic curves on fixed-function GPUs.

There are also other cases where segments can not be rasterized directly. The following list shows all the occasions where a segment can not be directly rasterized using my implementation of Loop and Blinn's approach:

- Cubic curve is self-intersecting. (See chapter 2.10.2 under Category 2: The Loop.)
- Cubic segment is really a quadratic, a line or a point. (See 2.10.2 under Category 4 and 5.)
- Cubic segment on fixed-function platform. (See chapter 5.2.3.)
- Cubic segment with concave control polygon. (See chapter 7.3.2.)
- Segment will cause rasterization errors due to limited precision in GPU (See chapter 5.2.5)

All these cases can however be easily solved by subdividing the offending segment until it can be rasterized and/or by using a simpler segment type as an approximation.

The method ApproximateWithRasterizable generates a list of rasterizable segments that approximate the original segment. The total error in pixel units, including the maximum distance between the approximation curve and the original segment, is guaranteed to be less than maxDistance. The method is abstract in the Segment class and is overloaded by all the segment types. (See figure 6.1.)

OpenVG supports the following segment types: (It also supports some other types that can be trivially converted to one of the below)

- Line
- Quadratic curve
- Cubic curve
- Elliptical arc

Segment of a type that cannot be rasterized directly are subdivided or approximated with one that can be rasterized directly:

- Lines can always be rasterized directly. (No action is needed - the interior polygon alone gives the desired rasterization result.)
- Quadratic curves are subdivided until they have small enough rasterization error that they can be rasterized directly.
- Cubic curves are subdivided until they can be either approximated by a quadratic curve with small enough total error, or rasterized directly.
- Elliptical arcs are directly rasterized if the rasterization error is small enough. If not, it is subdivided until it can be approximated with quadratic curves with small enough total error. (Subdivision does not reduce rasterization error for elliptical arcs.)

I will now describe how ApproximateWithRasterizable works for all of the segment types that are supported by OpenVG.

Listing 6.2: Line::ApproximateWithRasterizable - Approximate line with rasterizable segments

```
Segment[] Line::ApproximateWithRasterizable()
{
    // Line segment is always directly rasterizable. Return list with one
       element: this.
    return Segment[](*this);
}
```

The pseudocode implements a polymorphic method ApproximateWithRasterizable for all the supported segment classes. The method generates and returns a rasterizable approximation path in the form of an array of Segments.

My descriptions include both simple informal pseudocode and C++-like pseudocode. In my informal pseudocode I will not take into account the functionality to emulate a traditional polygonal approximation algorithm. I will also not explain exactly which approximation, rasterization and other errors that must be calculated at each step.

The C++-like pseudocode will however include all this functionality. The global functions int GetMaxDegree() and int CanRenderEllipticalArcs() are used for emulating a traditional polygonal approximation implementation. They control which segment types are allowed to be rasterized directly: GetMaxDegree() selects the maximum degree of Bézier curves that are supported. (Degree 1 is a line, degree 2 a quadratic, degree 3 a cubic.) CanRenderEllipticalArcs() determines whether direct rasterization of elliptical arcs is supported.

6.2.3 About the maxSnapError constant

Error does not only come from simplified geometry (approximationError) and limited fragment shader precision (rasterizationError). The maxSnapError constant is an error bound for implicit error introduced by the GPU.

Because the GPU's rasterizer usually operates in fixed point, vertices are snapped to a fine grid before rendering. In a real-world OpenVG implementation, this must be accounted for. This can be done by including the worst error that could result from this snapping in the comparison against maxDistance. The error *maxSnapError* is half of the diagonal distance between nodes in the fine grid - the maximum error that can be introduced by snapping to the nearest node. I will assume a fine grid with 16x16 nodes per pixel. This gives a maxSnapError of $\frac{1}{2}\sqrt{\frac{1}{16}^2 + \frac{1}{16}^2} = 0.04419$.

Line

Lines do not need any special handling. The interior polygon correctly produces the desired straight edge that is the purpose of this segment type. See listing 6.2 for C++-like pseudocode for ApproximateWithRasterizable - it simply returns an array with copy of itself.

Quadratic Curve

Quadratic curves can not always be rasterized directly due to limited internal GPU precision. In this case, they must be subdivided.

When a quadratic segment is very close to a line, geometry can be simplified by rasterizing the segment as a line. Rasterizing lines is cheaper than rasterizing quadratic curves, mainly because it saves one triangle. It is therefore beneficial to try and approximate quadratic curves with lines when possible.

The procedure for approximating a quadratic curve with rasterizable segments is as follows:

1. Is a line approximation acceptable within the total error threshold?
    - Yes: Return from the function with a line segment approximation.

2. Can the quadratic segment be rasterized directly?
    - Yes: Return from the function with a copy of itself.
3. Split the segment at the middle into two sub-segments.
4. Recurse into both sub-segments and return from the function with the results combined into a single list.

See listing 6.3 for C++-like pseudocode implementing this functionality.

Approximation with lines can destroy the smooth look of the curves, so a lower error threshold may be desirable for this purpose. See chapter 10.2.9 for a discussion.

Cubic Curve

Cubic curves can not always be rasterized directly due to limited internal GPU precision. In this case, they must be subdivided.

When a cubic segment is very close to a line, geometry can be simplified by rasterizing the segment as a line. Rasterizing lines is cheaper than rasterizing cubic curves, mainly because it saves one triangle. It is therefore beneficial to try and approximate quadratic curves with lines when possible. Similarly, cubic curves can often be approximated with quadratic curves.

Loop and Blinn's approach requires special case handling when the curve is close to a quadratic curve, a line or a point. I handle this here by always testing whether the curve can be approximated with a quadratic curve within the specified rasterization error bound using maxDistance. If it can not, I assume that it is safe to rasterize it directly.

Self-intersections can occur in the cubic curve rendering algorithm, in which case the curve must also be subdivided. Concave control polygons are also not supported in my current implementation.

The procedure for approximating a cubic curve with rasterizable segments is as follows:

1. Is a line approximation acceptable within the total error threshold?
    - Yes: Return from the function with a line segment approximation.
2. Is a quadratic approximation acceptable within the total error threshold?
    - Yes: Return from the function with a quadratic segment approximation.
3. Is direct rasterization of cubic curves supported on this platform? (fixed-function vs. programmable)
    (a) Is the cubic curve self-intersecting outside 0<t<1?
        - Yes: Subdivide at the offending parameter value and jump to 5.
    (b) Can the cubic segment be rasterized directly? (Check total error and whether the control polygon is convex)
        - Yes: Return from the function with a copy of itself.
4. Split the segment at the middle into two sub-segments.
5. Recurse into both sub-segments and return from the function with the results combined into a single list.

See listing 6.4 for C++-like pseudocode implementing this functionality.

Approximation with lines can destroy the smooth look of the curves, so a lower error threshold may be desirable for this purpose. Approximation with quadratic curves however looks much better, since the smoothness of the curve is not ruined. See chapter 10.2.9 for a discussion.

Listing 6.3: Quadratic::ApproximateWithRasterizable - Approximate quadratic curve with rasterizable segments

```
// Calculate position at parameter value t
Vector2 Quadratic::GetPosition(float t);

// Split the quadratic curve into two quadratic curves at the specified parameter value. Return a list of the two segments.
Segment[] Quadratic::Subdivide(float t);

// Approximate a quadratic curve using a line. Return the approximation error bound.
float Quadratic::ApproximateWithLine(Line* approx);

const float maxDistanceForLineApprox = maxDistance;

Segment[] Quadratic::ApproximateWithRasterizable()
{
    // Try to approximate with a line
    {
        Line approx;
        float lineError = maxSnapError + ApproximateWithLine(&approx);
        if (lineError < maxDistanceForLineApprox) {
            return Segment[](approx);
        }
    }

    // Try to rasterize the quadratic curve directly.
    if (GetMaxDegree()>=2) {
        // See if I have enough precision to rasterize directly using Loop and Blinn's approach.
        float quadError = maxSnapError + GetMaxRasterizationError();
        if (quadError < maxDistance) {
            return Segment[](*this);
        }
    }

    // The approximation or direct rasterization was not possible. I must subdivide the segment and try again. This is done by subdividing and recursing.
    Segment[] subPaths = this.Subdivide(0.5);
    Segment[] approx_l = subPaths[0].ApproximateWithRasterizable();
    Segment[] approx_r = subPaths[1].ApproximateWithRasterizable();
    return Segment[](approx_l, approx_r);
}
```

Listing 6.4: Cubic::ApproximateWithRasterizable - Approximate cubic curve with rasterizable segments

```
// Approximate cubic curve using a line.
float Cubic::ApproximateWithLine(Line* approx);

// Approximate cubic curve using a quadratic. Return the approximation error bound.
float Cubic::ApproximateWithQuadratic(Quadratic* approx);

// Test that none of the illegal conditions for cubic curves are present
bool Cubic::IsRasterizable();

const float maxDistanceForLineApprox = maxDistance; // should maybe be lower

Segment[] Cubic::ApproximateWithRasterizable()
{
    // Try to approximate using a line.
    Line approx;
    float lineError = maxSnapError + ApproximateWithLine(&approx);
    if (lineError < maxDistanceForLineApprox) {
        // error was small enough: return approximation
        return Segment[](approx);
    }

    // Can I rasterize quadratic curves directly?
    if (GetMaxDegree() >= 2) {
        // Can I approximate with quadratic?
        Quadratic approx;
        float quadraticError = maxSnapError + ApproximateWithQuadratic(&approx);
        if ( quadraticError < maxDistance) quadraticError += approx.GetMaxRasterizationError();
        if ( quadraticError < maxDistance ) {
            // error was small enough: return approximation
            return Segment[](approx);
        }
    }

    // Can I rasterize the cubic curve directly?
    if (GetMaxDegree()==3) {
        // Can I rasterize the segment directly using loop and blinn's approach?
        if ( IsRasterizable() ) {
            // Do I have good enough precision?
            float cubicError = maxSnapError + GetMaxRasterizationError();
            if ( cubicError < maxDistance ) {
                return Segment[](*this);
            }
        }
    }

    // The attempts to approximate or directly rasterize this cubic curve have failed. This is solved by subdividing and recursing.
    Segment[] subPaths = Subdivide(0.5);
    Segment[] approx_l = subPaths[0].ApproximateWithRasterizable();
    Segment[] approx_r = subPaths[1].ApproximateWithRasterizable();
    return Segment[](approx_l, approx_r);
}
```

Elliptical Arcs

Elliptical arcs are handled much like quadratic curves, but with an extra special case. Unlike with Bézier curves, rasterization error of elliptical arcs can not be decreased by subdivision.

The main function for approximating an elliptical arc with rasterizable segments works as follows:

1. Can the elliptical arc be rasterized directly? (Check total error)
    - Yes: Return from the function with a copy of itself.
    - No: Call a function which approximates the elliptical arc using quadratic curves and return the result.

That is, a check is performed to see if the elliptical arc can be directly rasterized. If so, a copy is returned. If not, a second, recursive function is called. It works as follows:

1. Is a line approximation acceptable within the total error threshold?
    - Yes: Return from the function with a line segment approximation.
2. Can the quadratic approximation acceptable within the total error threshold?
    - Yes: Return from the function with a quadratic approximation.
3. Split the segment at the middle into two sub-segments.
4. Recurse into both sub-segments and return from the function with the results combined into a single list.

See listing 6.5 for C++-like pseudocode implementing this functionality.

6.2.4 Approximation and Approximation Error

The polymorphic methods ApproximateWithLine and ApproximateWithQuadratic methods in the pseudocode above converts a segment into a simpler segment type. They return an error bound in pixel units for the distance between the original curve and the approximation. I need to be able to convert from any type to line, and from cubic and elliptical arc to quadratic.

A feature of my path rasterizer is that it can be turned into a traditional polygonal approximation rasterizer. For this purpose, all segment types have the ability to perform approximation with line segments.

Conversion From Quadratic to Line (Quadratic::ApproximateWithLine)

A line approximation of a quadratic curve can be created by simply connecting its starting point to its end point.

An approximation error bound can be found by taking the distance from the midpoint of the quadratic curve to the baseline. The midpoint is found by evaluating the parametric equation for quadratic curves at $t = 0.5$. An informal proof that this is in fact an error bound is given here:

Consider the explicit equation $y(x) = x^2$ for the canonical curve, found in chapter 5.2.2. The curve is furthest from the baseline at x=0, which is the middle of the curve and corresponds to t=0.5 in the parametric representation. Since all quadratic curves can be represented by an affine transformation of this canonical curve, no piece of the curve can ever be further away from the baseline than $t = 0.5$.

The method Quadratic::ApproximateWithLine tries to approximate a quadratic curve using a line. It returns an upper bound for the distance between the original curve and the approximation. C++-like pseudocode is given in listing 6.6.

Listing 6.5: EllipticalArc::ApproximateWithRasterizable - Approximate elliptical arc with rasterizable segments

```
1
2  // Approximate elliptical arc using a line. Return the approximation error
       bound.
3  float EllipticalArc::ApproximateWithLine(Line* approx);
4
5  // Approximate elliptical arc using a line. Return the approximation error
       bound.
6  float EllipticalArc::ApproximateWithQuadratic(Quadratic* approx);
7
8  // This private method is used when the elliptical arc cannot be rasterized
       directly, and resort to approximation with line or quadratic, depending
       on the GetMaxDegree().
9  Segment[] EllipticalArc::ApproximateWithRasterizableQuadratics()
10 {
11     if (GetMaxDegree()==1) {
12         Line approx;
13         float lineError = maxSnapError + ApproximateWithLine(&approx);
14         if (lineError < maxDistance) {
15             return Segment[](approx);
16         }
17     }
18     else // GetMaxDegree is at least 2.
19     {
20         Quadratic approx;
21         float quadError = maxSnapError + ApproximateWithQuadratic(&approx);
22         if ( quadError < maxDistance ) quadError +=
               GetMaxRasterizationError();
23         if ( quadError < maxDistance ) {
24             return Segment[](approx);
25         }
26     }
27
28     // subdivide and recurse
29     Segment[] subPaths = Subdivide(0.5);
30     Segment[] approx_l = subPaths[0].ApproximateWithRasterizableQuadratics
           ();
31     Segment[] approx_r = subPaths[1].ApproximateWithRasterizableQuadratics
           ();
32     return Segment[](approx_l, approx_r);
33 }
34
35 Segment[] EllipticalArc::ApproximateWithRasterizable()
36 {
37     // Can I rasterize the elliptical arc directly?
38     if ( CanRenderEllipticalArcs() )
39     {
40         float ellipseError = maxSnapError + GetRasterizationError();
41         if (ellipseError < maxDistance) {
42             return Segment[](*this);
43         }
44     }
45
46     // Failed to rasterize the ellipse. Must approximate with quadratics.
47     return ApproximateWithRasterizableQuadratics();
48 }
```

Listing 6.6: Quadratic::ApproximateWithLine - Approximate quadratic curve with line segment

```
float Quadratic::ApproximateWithLine(Line* approx)
{
    // create the linear approximation
    approx.sp = this.sp;
    approx.ep = this.ep;

    // calculate the midpoint of the real curve
    Vector2 realMidpoint = GetPosition(0.5);

    // calculate rasterization error bound: the distance from the real
       curve's midpoint to the approximation baseline.
    return approx.DistanceToPoint( realMidpoint );
}
```

Listing 6.7: Cubic::ApproximateWithLine - Approximate cubic curve with line segment

```
float Cubic::ApproximateWithLine(Line* approx)
{
    // Create linear approximation trivially
    approx.sp = this.sp;
    approx.ep = this.ep;

    // Calculate the midpoint of the real curve
    Vector2 realMidpoint = GetPosition(0.5);

    // Calculate distances from the control points to the baseline
    float distance_cp0 = approx.DistanceToPoint( this.cp0 );
    float distance_cp1 = approx.DistanceToPoint( this.cp1 );

    // The maximum of those values is an upper bound to the approximation
       error
    return max(distance_cp0, distance_cp1);
}
```

Conversion From Cubic to Line (Cubic::ApproximateWithLine)

A line approximation of a cubic curve can be created by simply connecting its starting point to its end point.

An approximation error bound can be found by taking the largest of the distances from the control points of the cubic curve to the baseline. This conservative subdivision criterion is described in [25].

NOTE: This calculation approximation error bound is preliminary. While the approach described here will return a correct error bound, it is unnecessarily conservative and will therefore lead to more subdivision than required. This makes it unsuitable for benchmarking. Further research is required.

C++-like pseudocode is given in listing 6.7.

Conversion From Cubic to Quadratic (Cubic::ApproximateWithQuadratic)

Cubic curves must sometimes be converted to quadratic curves, for the reasons explained first in chapter 6.2.2.

The following approaches for creating a quadratic approximation and calculating the error bound are much used and are described among other places in [26]. However, I have not been able to find an evidence that the method is robust in all cases.

Listing 6.8: Cubic::ApproximateWithQuadratic - Approximate cubic curve with a quadratic curve

```
float Cubic::ApproximateWithQuadratic(Quadratic* approx)
{
    // Move start and end points in the direction of their normals (The
       normals at the start and end points equal the normals of the control
        polygon edges)
    approx.sp = sp;
    approx.ep = ep;

    // calculate tangents at start and end point of cubic curve
    Vector2 sp_tangent = cp0-sp;
    Vector2 ep_tangent = ep-cp1;

    // Find intersection of the tangents at start and end point of cubic
       curve (calculations are omitted)
    approx.cp = ...

    // Get the midpoint of the approximated curve
    Vector2 approxMidpoint = approx.GetPosition(0.5);

    // Get the midpoint of the real curve
    Vector2 realMidpoint = this.GetPosition(0.5);

    // Calculate rasterization error bound: the distance between the
       midpoint of the real and the approximated curve
    float maxApproxError = (realMidpoint-approxMidpoint).GetMagnitude();

    // Return the approximation error bound. It will be added to later
       error comparisons and may lead to the segment being subdivided.
    return maxApproxError;
}
```

A traditional method for creating a quadratic approximation of a cubic curve is to use the intersection of the tangents at the start and end points as control point. This preserves the tangents at the start and end points of the approximation so that the path will look smooth even with a rather high error bound.

A conservative approximation error bound is found by taking the distance between the midpoint of the cubic and the quadratic curve. However, I have not found evidence that this method is robust - and never returns a too low value.

When the tangents are almost parallel, a division with zero will occur and the control point will be placed at infinity. This gives the correct result as the error estimation will return infinity, and this will again lead to the segment being subdivided.

When a control point overlaps the start or end point, the tangent can become a null vector, and the approximation will fail to find a solution for the control point. This situation should be avoided.

The method Cubic::ApproximateWithQuadratic approximates a cubic curve with a quadratic curve. It returns an upper bound for the distance between the real curve and the approximation. C++-like pseudocode is given in 6.8.

NOTE: This method for converting from cubic to quadratic curves is preliminary. Although it is a common approach, it has some problems:

First, it is conservative and may thus lead to unnecessary subdivision. Second, I have not found evidence that the method is robust. If the method is not robust and sometimes returns a too low approximation error, insufficient subdivision will occur and an implementation will not conform to the OpenVG specification.

Future work is needed to make sure that the error estimation method used never gives a too low value for the approximation error. A less conservative error calculation may also be beneficial.

Conversion From Elliptical Arc to Line (EllipticalArc::ApproximateWithLine)

This method tries to approximate an elliptical arc using a line. It returns an upper bound for the distance between the real curve and the approximation.

Simply use the baseline as the approximation. It is created by connecting the start point and end point of the quadratic curve.

An equation for calculating the maximum error when approximating an ellipse with a line is given in [39].

Conversion From Elliptical Arc to Quadratic (EllipticalArc::ApproximateWithQuadratic)

An efficient and slightly conservative method for fitting an ellipse with a quadratic curve, including pseudocode, is given in [39].

The method EllipticalArc::ApproximateWithQuadratic tries to approximate an elliptical arc using a quadratic curve. It returns an upper bound for the distance between the real curve and the approximation. Implementation of this functionality is left for future work.

6.2.5 Support for All Paints and Blend Modes

The requirement specification requires that support for paints and blend modes must be implementable. The stencil algorithm used by the Kokojima approach is very convenient in that it separates the rasterization and the drawing into two different passes: In the rasterization stage, the path is rendered into the stencil buffer, but the color buffer is not updated. The shape of the path is now represented by the values in the stencil buffer. Then, the drawing itself takes place by drawing a full-screen quad and using stencil tests to kill pixels that are outside the path. Thus, the drawing itself is completely separated from the rasterizing.

My approach only occupies the stencil buffer, or at least 8 bits of it. The remaining bits can still be used. If even more stencil bits are required, the fill rule can be evaluated in an additional pass after rasterization and before the drawing pass to reduce the number of bits in use to 1. Beyond these stencil bits there is no limitation to what GPU features can be used by the paint and blend modes. The problem of drawing a path with correct paint and blending can thus safely be left for a later project without fear that my path rasterizer will interfere with that algorithm.

For the purpose of this report, I will draw using simple single-color paint.

6.2.6 Stroking

Stroking means to paint the outline of a path as if traced by a pen. I will perform stroking by generating a path that represents the area to be filled. It will consist of multiple subpaths that are allowed to overlap since this greatly simplifies generation of the subpaths. The result is rendered using the non-zero fill rule, which means that the overlap is not visible.

The OpenVG specification requires support for multiple types of join and cap styles including flat, butt, miter and round. To support this, one subpath is created for each join and cap.

Stroke geometry consists of multiple subpaths generated individually from each input segment. For each segment, two offset curves are calculated: One for the inside and one for the outside of the stroke. They are connected by one line segment at the start and one at the end of the segment, forming flat endings in the direction of the curve normal. See chapter 2.9 for an explanation of offset curves.

Figure 6.2 shows how the final stroke shape is drawn using overlapping subpaths for joins, caps and stroke geometry.

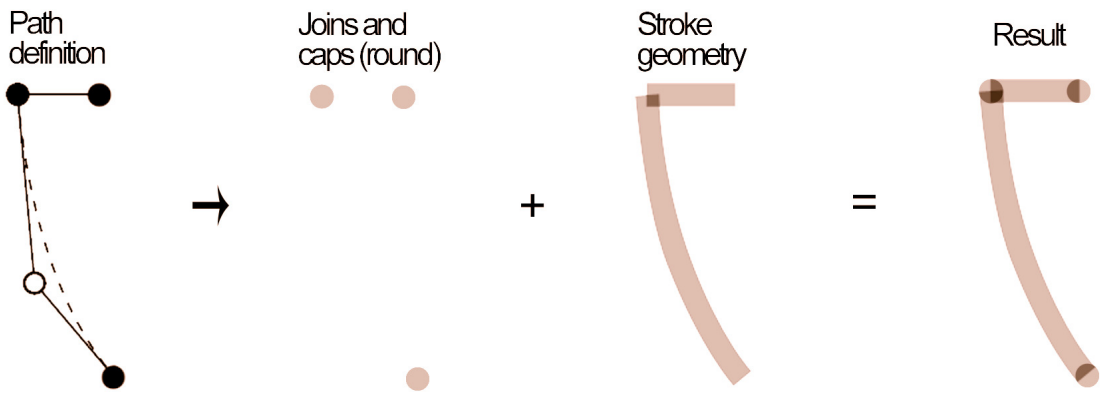


Figure 6.2: Subpaths for joins, caps and stroke geometry.

Creating Offset Curve of Quadratic

As an example on how to create an offset curve, I have included pseudocode for creating offset curves of quadratic curves. They are based on Tiller and Hanson's approach [47]. See listings 6.9 and 6.10. Similar approaches for cubic curves and elliptical arcs can also be used for stroke generation, but the error estimation method must be replaced.

6.3 Prototype Implementation

This chapter describes the prototype implementation. I start by explaining how the implementation was approached guided by my requirement specification in chapter 6.3.1. I explain my choice of programming language and platform in chapter 6.3.2, and the choice of libraries in chapter 6.3.3. Finally, I provide an overview of the prototype's architecture in chapter 6.3.4.

6.3.1 Requirements

The requirement specification in my interpretation of the assignment (chapter) is the basis for a prototype that implements efficient OpenVG path rasterization. The specification defines the goal of the work in the long term rather than work that must be accomplished in this thesis. The whole specification has thus not been implemented.

The efficiency of the new approach should be measured and compared to a traditional approach. Thus, the prototype needs features that facilitate benchmarking.

Realistic test-sets should be used and are available from the OpenVG group at ARM Norway. The prototype needs to be able to load these data-sets. It is important that benchmarking is done using realistic data-sets so that the results are representable of real-world use.

Elliptical arcs seem to be in very little use, and I was not able to find any realistic high-pressure data-sets that used this kind of segment. I have explained how support for elliptical arcs can be implemented in much the same way as quadratic curves. They seem to be uncommon, and since I had problem finding any realistic test-sets with sufficient complexity, I chose not to provide an implementation of elliptical arcs. Lines, quadratic curves and cubic curve segments should be supported.

To facilitate benchmarking, the prototype should support a traditional approach to path rasterization. Recursive subdivision into lines followed by the stencil algorithm is classic, and is easily integrated with the algorithms used for efficient rasterization in the prototype.

Verification should be supported by providing a large number of synthetic tests that test all the features. There should be tests that use all supported segment types, subpaths and both fill rules with overlapping geometry. Additional informal testing is facilitated by allowing the user to manipulate the shapes by moving the control points interactively.

Stroking and dashing will not be implemented. Stroking was specified as optional in the project assign-

Listing 6.9: Offset curve approximation for quadratic curve

```
// Create a quadratic approximation of the offset curve of a quadratic.
// Parameter offset contains the distance for the offset curve.
// Returns the approximation error bound.
float Quadratic::CreateOffsetCurve(Quadratic* approx, float offset)
{
    // Move start and end points in the direction of their normals (The normals at the start and end points equal the normals of the control polygon edges)
    approx.sp = GetPosition(0.0) + GetNormal(0.0) * offset;
    approx.ep = GetPosition(1.0) + GetNormal(1.0) * offset;

    // the control point is then placed so that the shape of the control polygon, and thus the curve itself, is retained. (calculations are omitted)
    approx.cp = // ...

    // Get the midpoint of the approximated offset curve
    Vector2 approxMidpoint = approx.GetPosition(0.5);

    // Get the midpoint of the real offset curve
    Vector2 realMidpoint = this.GetPosition(0.5) + this.GetNormal(0.5) * offset;

    // Calculate rasterization error bound: the distance between the midpoint of the real and the approximated curve
    float maxApproxError = (realMidpoint-approxMidpoint).GetMagnitude();

    // Return the approximation error bound. It will be added to later error comparisons and may lead to the segment being subdivided.
    return maxApproxError;
}
```

Listing 6.10: Offset curve approximation for quadratic curve

```
// Create a linear approximation of the offset curve of a quadratic.
// Parameter offset contains the distance for the offset curve.
// Returns the approximation error bound.
float Quadratic::CreateOffsetCurve(Line* approx, float offset)
{
    // Create the approximation by extruding start and end point along their normal by offset pixel units
    approx.sp = GetPosition(0.0) + GetNormal(0.0) * offset;
    approx.ep = GetPosition(1.0) + GetNormal(1.0) * offset;

    // Calculate the midpoint of the real offset curve
    Vector2 realMidpoint = GetPosition(0.5) + GetNormal(0.5) * offset;

    // Calculate approximation error bound: the distance from the real offset curve's midpoint to the approximation line.
    return approx.DistanceToPoint( realMidpoint );
}
```

ment, and dashing was not explicitly mentioned. I have therefore chosen to focus on filling of paths rather than stroking. (A suitable approach to stroking is however described in chapter 6.2.6.)

6.3.2 Choice of Platform and Programming Language

The target platform for the assignment is to run on OpenGL ES 1.1 and 2.0-compatible GPUs. Such target platforms are not yet easily available and may be difficult to program for. Therefore, while the algorithms have been chosen and developed for handheld graphics devices, the prototype was developed for a desktop computer with the OpenGL 2.0 API.

OpenGL ES 1.1 and 2.0 are not currently available for desktop GPUs, but OpenGL 2.0 is essentially a superset of these APIs and has all the required features. I have restricted myself to only using features that are available in the target platform. Therefore, porting my implementation should be very easy.

It is expected that a hardware developer implementing OpenVG may instead want to use a proprietary low-level API. This may be beneficial for two reasons. First, a proprietary API can make it possible to optimize better for the target platform. Second, it is not possible to implement all the features of OpenVG on top of OpenGL ES 1.1, and there is question whether it is possible even on OpenGL ES 2.0 [40]. This problem can also be solved by exposing OpenGL ES extensions.

C++ was chosen as programming language. C is probably more common for drivers and on handheld devices, and is probably the preferred language for a low-level OpenVG implementation. However, C++ is preferred for the prototype since it is object-oriented and has more features. If used appropriately, this can make development faster and can simplify restructuring. These attributes are important when developing and experimenting with a prototype. OpenGL and OpenGL ES are based on C, and can be easily used from both C and C++.

6.3.3 Choice of Libraries

OpenGL 2.0 is used for rendering for the reasons described in chapter 6.3.2.

GLUT is used for creating a user interface and for window management. This choice was made because I knew that GLUT provided the necessary features at little overhead in lines of code. It has functionality to open a window for OpenGL output, print text overlays and capture mouse and keyboard input.

For easy access to OpenGL extensions, the Glee library is used. No more libraries were necessary.

6.3.4 Architecture Overview

The implementation is very similar to what is described in chapter 6.2. In addition to the classes listed there, the following classes are introduced: Path and Subpath. See figure 6.3 for updated UML diagram. Notice the following things:

1. Starting point is no longer stored in the Segment base class, but is passed down the call stack as a parameter. This removes redundant data.
2. Cubic has attributes t0, t1, t2, t3 that hold the evaluated varyings until rasterization.
3. EllipticalArcs are not supported

The application can be divided into the following parts:

1. Geometry classes: Path, Subpath, Poly, Segment and its children.
2. User Interface (ui.h/cpp corresponding namespace ui and main_glut.cpp)
3. Settings (settings.h/cpp corresponding namespace settings)
4. Collection of Statistics (Stats.h/cpp corresponding namespace stats)

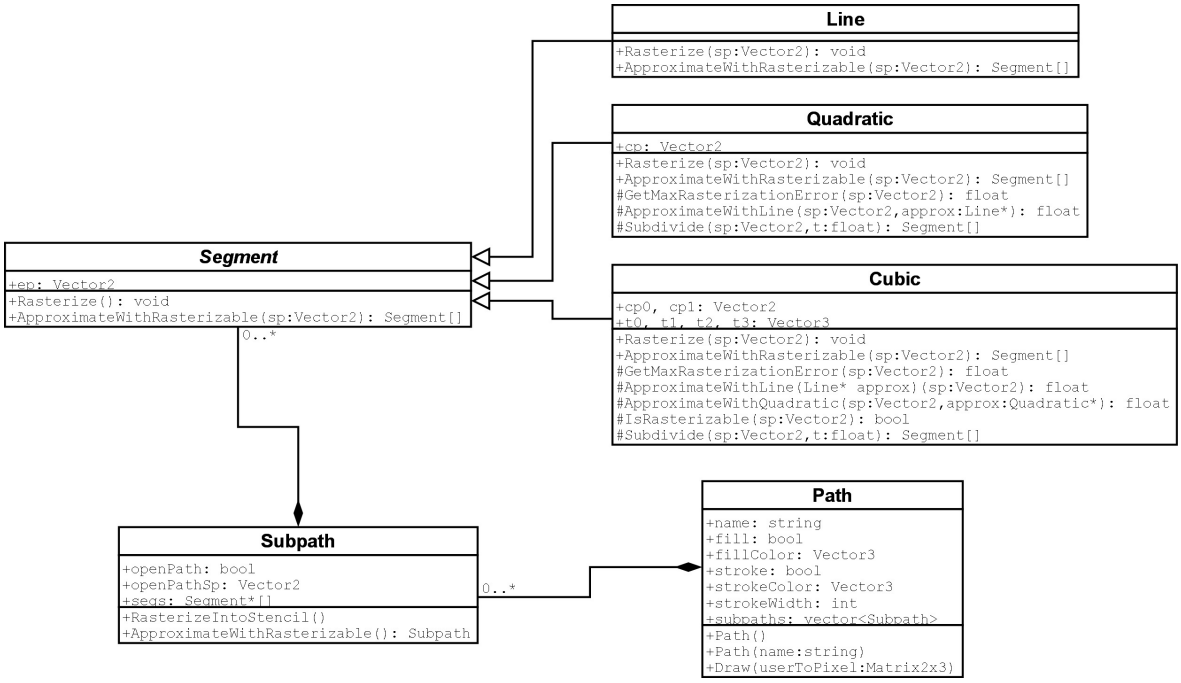


Figure 6.3: Class diagram of the prototype architecture.

5. Test-set Generation/Loading (testset.h/cpp)

6. Misc (shader_fw.h/cpp corresponding namespace shader_fw, vecmath.h/cpp, prec.h/cpp)

Geometry Classes

The segment classes are sufficiently explained in the previous chapter, specifically chapter 6.2.

The Path class represents a path. In addition to a vector of subpaths, its attributes include a name stored as a string and some settings that define how it is rendered. The only rendering settings that are actually used by the prototype are: fillRule and fillColor. In addition, if the fill attribute is false, nothing is drawn. The attributes are public, so they can be filled in directly. There is a method called Draw which is used to render the path. It takes a transformation matrix and a boolean called drawWireframe which is used for debug rendering.

The Subpath class contains mainly a list of segments. In addition to a list of pointers to segments, it has a boolean called openPath. If this is true, the path represents an open path, that is, a path which starts and ends at different places. There is an attribute of type Vector2 called openPathSp. For an open path, this defines the starting point of the path. For a closed path, the end point of the last segment is used as starting point, thus forcing a closed shape.

The method ApproximateWithRasterizable takes a transformation matrix as argument and generates a new subpath based on the current one, which is guaranteed to contain only rasterizable segments with a maximum error less than maxDistance.

For rasterizing the subpath into the stencil buffer, the method RasterizeIntoStencil is used.

The Poly class represents a polygon, and supports two different methods for rasterizing into the stencil buffer. The polygon is represented by the attribute points, a vector of Vector2 points. RenderSimpleFan rasterizes using the stencil algorithm and traditional triangle fan triangulation, while RenderDividingTriangle rasterizes with my dividing triangle triangulation approach. Its parameter openPath should be set if the first and last point of the polygon are assumed to be far from each other, such as the interior polygon resulting from a typical open path.

User Interface

The user interface is contained in the files main_glut.cpp and ui.h/cpp. The file main_glut.cpp contains the main function which is the entry point for the application, and all the glut call-back functions. Some

of the user interface actions are performed here (zoom, pan, exit) while others are forwarded to functions in the ui namespace through the ui::Action function. This file calls initialization functions and generates or loads a test-set into ui::paths before starting the glut main-loop.

The namespace ui contains the code for drawing the ui (the text and the path) and most of the code for executing commands from the user. This includes drag-and-drop of control points, changing rendering mode and switching active path. The actual path data for the application resides in the ui namespace in a vector of Path objects called ui::paths.

Settings

The settings namespace is used from everywhere and contains values that are supposed to be constants on one specific device. In the prototype however, it is interesting to adjust these values, and many of them can be modified through the user interface. The other ones can be changed by modifying the file settings.cpp. I will now explain what they do.

bool useProgrammablePipeline
Changes between fixed-function and programmable pipeline rendering. Can be changed from within the application by pressing r. The default value is true if the host GPU supports the programmable pipeline approach. If not, the default value is false.

float maxError
Specifies the maximum difference between the real path and the rasterized result. Should be 1.0 or less to conform with the OpenVG specification.

float maxSnapError
Specifies the inherent error from the GPU's rasterizer. The default value is the error generated by snapping vertices to a fine grid with 16x16 nodes per pixel.

int maxDegree
Corresponds to GetMaxDegree() in chapter 5. Specifies the maximum allowed degree of Bézier curves. 1=line, 2=quadratic, 3=cubic. This can be used for comparing the results of a traditional polygonal approximation with direct cubic segment rasterization. This value can be changed from within the application by pressing d. The default value is 3.

int pixelShaderMantissaBits
Specifies the internal precision in the target fragment shader unit. It is used for calculating rasterization error when using Loop and Blinn's approach for rasterizing quadratic and cubic curves. The default value is 10, which is the minimum precision required by the OpenGL ES Shader Language [12].

quadraticLutWidth and quadraticLutHeight
Specify the dimensions of the look-up texture used for rasterizing quadratic curves in the fixed-function pipeline approach. This can not be modified while the program is running. The default values are 256x256.

int tileDimension
This value is used as basis for the calculation of tile list command count. The default value is 16, which specifies that the GPU uses tiles with 16x16 pixels.

bool useDividingTriangle
Decides whether the dividing triangle or triangle fan triangulation approach is used for the stencil algorithm when rasterizing interior polygons. The default value is true. It can be changed from within the application by pressing t.

Collection of Statistics

Functions for collection of statistics are found in the files Stats.h/cpp. The class Stats contains data that can be collected over a period of time.

The namespace stats contains tools for collecting data for a single frame. By calling the functions NewTriangle and NewSegment, statistics about the geometry is collected. At the end of the frame, a call

to PrintFrameStats prints statistics to the console window. Before collecting new statistics for the next frame, a call to NewFrame will reset the counters in stats::frame.

Test-Set Generation/Loading

The files testset.cpp/h contain tools for loading and generating paths intended for testing the prototype. The h-file exposes two functions:
GenerateTestset and LoadTestset.

Each of them takes a reference to a vector of paths. They fill this vector with test-data. GenerateTestset generates some synthetic tests. The resulting paths are designed to be viewed one at a time. LoadTestset loads a file in path format specified by the filename parameter. (See the comment about the path format in chapter A.1.)

Misc

Some functions related to uploading of shader programs to OpenGL are in the namespace shader_fw, in files shader_fw.h/cpp.

Classes for vector maths, specifically Vector2, Vector3 and Matrix2x3 are defined globally in vec-math.h/cpp.

A function called GetErrorAlongEdge for calculating error due to interpolation of a varying along an edge is found in the files prec.h/cpp. This function is discussed and explained in chapter 5.2.5.

6.4 Summary

I have described an approach which efficiently fills the interior of paths using both fill rules and with a guaranteed error bound. It supports both fixed-function and programmable GPUs. Paths may consist of lines, quadratic curves and cubic curves. An approach for rendering elliptical arcs is partially described but not implemented. Stroking and dashing is also partially described but not implemented.

The algorithms used promise significant improvement over traditional polygonal approximation and tessellation.

7 Prototype Verification

The prototype implementation verification process is described in this chapter. Although verification was specified as optional in the project assignment text, functional verification is performed, while verification of rasterization error is left for future work.

The official OpenVG conformance test suite is discussed in chapter 7.1. A discussion of the test method is provided in chapter 7.2. The functional verification process is performed in chapter 7.3 and a specific bug is discussed. Verification of maximum rasterization error is discussed in chapter 7.4, but actual verification is left for future work.

7.1 About OpenVG Conformance Tests

An official OpenVG conformance test suite is currently being developed by members of the Khronos Group. It runs a large number of very difficult special cases of curves and similar. Conformant renderers are required to correctly rasterize these difficult paths. An obvious approach would be to use this test suite for verification of my implementation.

There are a number of difficulties related to using the conformance test suite for verification of the prototype:

The test suite is implemented using OpenVG. A partial OpenVG implementation needs to be in place to use it. Also, the OpenVG conformance test suite is not open to the public.

Using the conformance suite to test the prototype is suggested for future work in chapter 10.2.1.

7.2 Method

The verification of my prototype will be divided in two tasks:

- Functional Verification
- Rasterization Error Verification

I will not test for bugs in the user interface, test-set loader and similar, since this is not the main focus of the assignment. These components only need to be bug-free enough that verification, testing and benchmarking of the rasterizer itself can be performed. Only the path rendering component will be targeted by the tests.

Functional verification uses a test plan with synthetic test cases to verify that difficult cases produce the correct result. This is done in chapter 7.3

Rasterization error tests are performed with synthetic and realistic test-sets. I have not found time to perform thorough testing, as it is specified as optional in the assignment. However, I have described how rasterization error tests could be done in chapter 7.4.

7.3 Functional Verification

The main purpose of my implementation is to prove that my approach produces correct output, and to facilitate benchmarking. It is not essential that it passes all functionality tests, as long as the reason can be explained and it can be used for benchmarking. However, there are two good reasons to perform at least some functionality testing:

- To locate and fix major bugs that pollute benchmarking results.
- To find problems with the algorithms.

Errors should be investigated to decide whether it is a problem with the implementation or the approach itself. If the problem is in the implementation, a decision must be made whether the bug should be fixed or left. If a test gives the wrong result, but will not affect benchmarking, and is difficult to fix, it can be left as it is. That said, I am not aware of any bugs in my implementation.

Functionality is tested using a test-plan. The test procedure as well as the desired result is specified in advance before the testing is performed. Functional verification

7.3.1 Preliminary Test Plan

This chapter contains the preliminary test plan as well as test results. Further testing is left for future work.

Note: If the plan says that some parameters should be *varied*, this means to try all the combinations. Default values should be used for unspecified parameters.

Functionality under test: Polygon, Quadratic and Cubic Rendering

Test-set: Cubic Shape
Render Mode Settings: Vary maximum degree
Expected Result: Render as expected
Result: Success

Functionality under test: Dividing Triangle Visual Result

Test-set: tiger.path
Render Mode Settings: Vary triangulation algorithm
Expected Result: Render as expected. Same visual result for all render modes
Result: Success

Functionality under test: Dividing Triangle Triangulation Result

Test-set: tiger.path
Render Mode Settings: Wireframe mode. Vary triangulation algorithm.
Expected Result: With dividing triangle algorithm, tile list command count should be lower, while triangle count stays about the same
Result: Success

Functionality under test: Cubic With Concave Control Polygon

Test-set: Synthetic test "Cubic Curve" - move a control point to create a concave corner.
Render Mode Settings: Vary pipeline mode, curve degree
Expected Result: Render as expected. Same visual result for all render modes
Result: Success (But see 7.3.2)

Functionality under test: Self-intersecting Cubic

Test-set: Synthetic test "Cubic Curve" - move one control point below the baseline
Render Mode Settings: Vary pipeline mode, curve degree
Expected Result: Render as expected. Same visual result for all render modes
Result: Success (But see 7.3.2)

Functionality under test: Cubic with loop

Test-set: Synthetic test "Cubic Curve" - move control points to create a loop
Render Mode Settings: Vary pipeline mode, curve degree
Expected Result: Render as expected. Same visual result for all render modes
Result: Success

Functionality under test: Fill rules 1

Test-set: Synthetic test "Quadratic Shape"
Render Mode Settings: Vary fill rule
Expected Result: Outer shape should have a hole with both fill rules
Result: Success

Functionality under test: Fill rules 2

Test-set: Synthetic test "Quadratic Shape" - flip the winding of the innermost shape by mirroring it.
Render Mode Settings: Vary fill rule
Expected Result: Inner shape should be visible as a hole with odd/even, but not with non-zero fill rule
Result: Success

Informal testing has been performed in addition to the above tests. Mainly, I have played with the synthetic and realistic data-sets by dragging the control points around to verify that rasterization occurs as expected.

I have found one bug that is worth discussing since it indicates a possible problem in Loop and Blinn's paper. The bug was not found while executing the test plan, but I have included tests that would detect the bug if it reappears. See chapter 7.3.2 for a discussion of the bug.

There are currently no known bugs in the implementation.

7.3.2 Incorrect Rasterization of Segments With Concave Control Polygons

A test corresponding to "Cubic With Concave Control Polygon" above revealed that my implementation of Loop and Blinn's cubic curve rendering algorithm did not work correctly when the control polygon was concave.

I did not find any explanation in [37] of how to generate the bounding geometry for the curve when the control polygon is not convex, and had therefore implemented an algorithm that took the convex hull of the control polygon. This produced wrong visual results. That is, if the line from the starting point through the control points ending at the end point had one concave and one convex corner, the output looked wrong. Image a in figure 7.1 shows the expected result, and image b shows how it looks with the buggy version of the renderer.

See chapter 5.2.4 for an explanation of how the problem was avoided in the prototype by subdividing the segment.

7.4 Maximum Rasterization Error Verification

According to chapter 5.2.5, our algorithm should guarantee that the rasterized result of the implementation never deviates as much as maxDistance pixel units from the correct path. The tests described in this

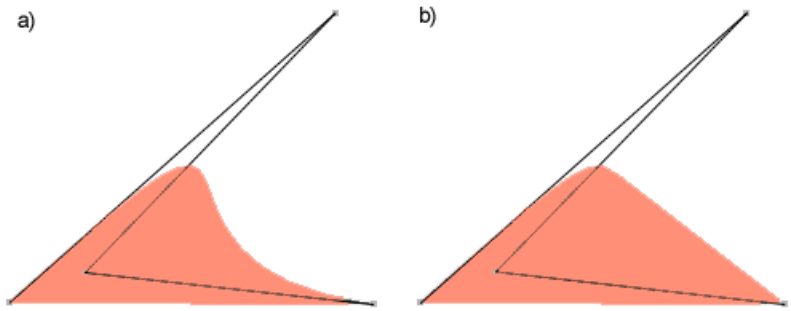


Figure 7.1: Incorrect rasterization of segments with concave boundary polygons.

chapter are designed to check that the rasterization error is according to the OpenVG specification.

For testing, I will render synthetic and realistic test-sets using different rendering modes and maxDistance $= 1.0$. The result will be compared to renderings produced by the OpenVG Reference Implementation (RI) supplied by the Khronos Group. The RI rasterizes with very high accuracy. Thus, the borders of my implementation's rasterization should always hit either an adjacent or the same pixel as the RI.

To verify the result, I will perform a subtraction operation between the result produced by my implementation and the reference implementation. Automatic testing is difficult, so the result will be studied visually to see that the result is as expected.

Suggestions for test-sets:

- Synthetic tests: All of the auto-generated tests (They are simple to verify)
- Realistic tests: tiger.path, dude.path and a test with quadratic curves

8 Benchmark Results and Discussion

This chapter describes how statistics from the prototype are collected for both the traditional polygonal approximation approach and the new, efficient path rasterization approach. The results are compared and discussed. The actual numbers from the collected statistics can be found in appendix B.

The method used for benchmarking is discussed in chapter 8.1. Some limitations of the method are discussed in 8.2. A presentation and discussion of the benchmark cases is provided in chapter 8.3, while the results are discussed in chapter 8.4.

8.1 Method

My implementation supports path rasterization using traditional as well as novel approaches. Being able to switch between the different rasterization techniques simplifies comparison between the new and the traditional techniques. Also, it ensures that the benchmarks are performed with identical thresholds and premises such as resolution and internal precision. I also have full control of all techniques in use so that there for example is no hidden caching that pollutes the benchmark results.

Test-sets are run using different rendering modes and settings. The number of rasterizable segments, polygons, vertices and tile list commands are collected and compared.

The decision of which statistics to collect and interpretation of the results is based on the discussion in chapter 4.1 about the relevance of various statistics.

No attempt is made to measure rendering time for the reasons discussed in chapter 4.1. The prototype in its current form is completely unoptimized and I have little understanding of where the bottlenecks are. Most of the time is probably spent allocating memory, performing redundant copying of data and similar avoidable tasks. It is my belief that these things would pollute results to an extent where they would only be misleading. Also, the approach is meant to run on a handheld device. Performance measured on a desktop computer would not be entirely representative.

It is not known exactly how much impact the various parameters of the benchmarking statistics have on performance as this will vary from device to device. Knowledge about the target platform must therefore be used estimate how fast a data-set will rasterize on a given device based on the number of polygons, tile-list commands etc. This can help in focusing further development of optimization towards a specific device.

The number of rasterizable segment commands in the result compared to the number of segments in the original path represents the amount of subdivision. This is the only non-linear algorithm that the CPU needs to perform. The time spent on other tasks are linearly dependent on the number of rasterizable segments that result from this task, and the number is therefore representable for the amount of CPU processing that must be performed. When it is equal to the number of segments in the original, the algorithm runs in linear time on the CPU. However, the new approach has a higher constant overhead per segment, and the numbers must be considered with this in mind.

I calculate the *subdivision overhead* as r/o, where r is the number of rasterizable segments, while o is the number of original segments. Thus, the subdivision can never be less than 100%, which means that

the algorithm runs in linear time.

The number of vertices, polygons and tile list commands affects the amount of memory traffic. Vertices and polygons must be written by the CPU to memory that can be accessed by the GPU. Tile list commands are written to memory and then read back by the GPU during rendering. The tile list command count is also an indication of how well the render target contents can be cached in an immediate mode renderer.

The new approach attempts to reduce all the measured statistics. *Improvement* is therefore calculated as $1-n/o$, where n is the new number and o is the old number, which gives a result between 0% and 100%. Note that calculating this value based on the subdivision overhead percentages is equivalent to using the same formula for the new and old rasterizable segment counts.

Note that what I call the *triangle fan* or *traditional* approach of triangulation is to draw a triangle fan towards the centroid of the polygon. This is more optimal than the naive method explained in most descriptions of the stencil algorithm, where the fan is drawn towards an arbitrary vertex of the polygon. Although this avoids one extra vertex and triangle, it is worse with regard to triangle shapes.

8.2 Limitations

Due to lack of time, the implementation has some issues that can potentially pollute the results: Error estimations used for approximation of cubic curves are fairly conservative, and may lead to more subdivision than necessary. I do not believe that this should not distort the statistics to a significant degree. Another and more serious problem is that concave control polygons are handled naively by subdivision. This reduces the benefit of the new approach. I will perform tests where I deactivate subdivision due to this reason to measure the severity of this issue. However, the results will not be completely accurate since bounding polygons are note accurately calculated for the cubic segments. I do however expect the benchmarking results to be nearly correct in this case.

More extensive benchmarking such as comparing statistics with other software packages is desirable. This is left for future work.

8.3 Benchmarks

All tests are run at a resolution of 512x512, a quadratic lut texture of 256x256 and a maxDistance of 1.0. Tile list command count is calculated for a tile based renderer with bounding box tiling and a tile size of 16x16 pixels. Although this statistic is calculated for a very specific hardware configuration, it is of some value for other forms of tile based and immediate mode GPUs as well. A high tile list command count compared to polygon count indicates a high amount of large and overlapping or sliver triangles. This is bad for all these kinds of renderers.

GPU floating point precision is specified as follows in the statistics:

- FP16: Floating point format with 10 mantissa bits
- FP24: Floating point format with 16 mantissa bits
- Great Precision: Imaginary floating point format with 100.000 mantissa bits

When referring to *the best configuration* I do not consider configurations with Great Precision, as these are imaginary configurations. I also do not consider tests with subdivision due to concave control polygon turned off when referring to the best configuration.

The test-sets were picked by me to form a representative selection of common use cases. For each test, I specify one common use case that I believe it is representative for. Note that this is my own opinion.

What follows now is a description of each test and a discussion of the result.

Please see appendix B for screenshots and statistics for the benchmark test cases.

8.3.1 Benchmark Case 1: Cubic Dude

The Cubic Dude (fyr.path) is a simple illustration consisting of 46 cubic curves. It was drawn in Adobe Illustrator, exported as SVG and converted to the .path format. This data-set is representative for simple illustrations based on cubic curves, made with programs such as Adobe Illustrator.

The stats improve steadily for each test configuration. The best results are achieved for rasterization with cubic curves with FP24. (Great Precision gives the same results) The difference between traditional polygon approximation and this configuration is shown in table 8.3.1.

Table 8.1: Measured improvements for Cubic Dude. Polygonal vs. FP24.

Statistic	From	To	Improvement
Subdivision Overhead	730%	187%	74.4%
Triangle Count	676	380	44.8%
Vertex Count	347	201	42.0%
Tile List Command Count	564	302	46.5%

It is apparent that memory traffic due to both geometry and tile lists can be reduced by about 45%, and the rasterizable segment count by a factor of 75%. The subdivision overhead is 187% in the best configuration.

The dividing triangle algorithm reduces tile lists by 21.4% for the polygonal approximation but only 6.2% for the best configuration.

Some testing was performed to see how much the issue with concave control polygons (see chapter 5.2.4) affects the result. Under "Additional Tests" in appendix B, I have run two more test configurations for the Cubic Dude: Cubic curve rendering with FP24 and Great Precision give the same results. Table 8.3.1 lists the improvement over polygonal approximation when subdivision due to concave control polygons is not performed.

Table 8.2: Measured improvements for Cubic Dude. Polygonal vs. FP24 with concave CP support.

Statistic	From	To	Improvement
Subdivision Overhead	730%	124%	83%
Triangle Count	676	294	56.5%
Vertex Count	347	158	54.5%
Tile List Command Count	564	272	51.7%

It is apparent that the issue with concave control polygons limits performance. When this subdivision criterion is deactivated, subdivision overhead is at 124%. This means that a relatively small amount of subdivision is performed. Memory traffic due to both geometry and tile lists can be reduced by over 50% compared to the traditional approach.

8.3.2 Benchmark Case 2: Quadratic Guy

The Quadratic Guy is a simple illustration consisting of 53 quadratic curves. He was created by destroying the Cubic Dude with hacks, and moving the control points around in my prototype. This data-set is representative for simple illustrations based on quadratic curves. Illustrations made in Adobe Flash are based on quadratic curves.

The same results are achieved by all configurations except polygonal approximation, including fixed-function rendering. The difference between traditional polygon approximation and the best configuration is shown in table 8.3.2.

Subdivision is not performed with the new approach, and CPU processing is thus minimized. Memory traffic due to vertices, polygons and tile list commands can be reduced by over 65% compared to the polygonal approximation approach.

Table 8.3: Measured improvements for Quadratic Guy. Polygonal vs. fixed-function.

Statistic	From	To	Improvement
Subdivision Overhead	540%	100%	81.5%
Triangle Count	578	190	67.1%
Vertex Count	298	106	66.4%
Tile List Command Count	530	184	65.3%

Since the shape consists of only quadratic curves, the issue with concave control polygon can not occur and thus results are not distorted by this.

The dividing triangle algorithm reduces tile lists by 21.1% for the polygonal approximation and 8.9% for the best configuration.

8.3.3 Benchmark Case 3: Chinese Text

The Chinese Text (chinese.path) test-set consists of 72.332 quadratic curves and lines, and contains rather small geometry with a lot of detail. This data-set is representative for text with medium to small size.

Subdivision overhead is at 101.4% for polygonal approximation, and 100% at the more advanced configurations. Thus, the new approach does not improve the statistics at all for this test. Triangle and vertex count is the same as with polygonal approximation, while the tile list command count varies only slightly between configurations.

The Chinese Text test has a lot of small details that can be easily approximated without introducing much error. Some curved segments are used, but they are only slightly curved and can be easily approximated by lines. The polygonal approximation method therefore performs almost no subdivision.

Moreover, the dividing triangle method actually produces up to 4.1% more tile list commands than the traditional triangle fan approach in this test.

8.3.4 Benchmark Case 4: Tiger

The famous postscript Tiger is a complex illustration consisting of 2.043 segments, mainly cubic curves. The original can be found in the gostscript distribution. It is representative for complex illustrations based on cubic curves. These can be made with programs such as Adobe Illustrator.

The stats improve steadily for each test configuration. The difference between traditional polygon approximation and the best configuration is shown in table 8.3.4.

Table 8.4: Measured improvements for Tiger. Polygonal vs. FP24.

Statistic	From	To	Improvement
Subdivision Overhead	243%	145%	40.3%
Triangle Count	9,956	9,134	8.3%
Vertex Count	5,180	4,352	16.0%
Tile List Command Count	5,720	4,200	26.6%

Subdivision overhead is significantly better for the new algorithm, but there is no significant reduction in geometry. Loop and Blinn's approach generates triangles for drawing the curve itself, and the triangle count is therefore not reduced as much as the segment count. The tile list command count is significantly more reduced, but this is mostly due to the dividing triangle algorithm.

The Tiger has a lot of small details that can be easily approximated by lines without introducing much error. The polygonal approximation method therefore works relatively well for approximating the paths in this test.

The dividing triangle algorithm reduces tile lists by 31.4% for the polygonal approximation and 13.4% for the best configuration. Thus, the polygonal algorithm with dividing triangles actually produces 6.5% fewer tile list commands than the best configuration.

Since cubic curves are present, the issue with concave control polygon may distort the benchmarks. I

have not performed this test without this subdivision criterion. This is however done for the next test, which also uses tiger.path.

8.3.5 Benchmark Case 5: Tiger Zoom

This is the same as case 5, but zoomed in by pressing the + key 30 times. It is representative for rendering complex illustrations in high resolution, or complex but large geometry such as maps. Note that no culling is performed for geometry that is outside the viewing window, so the test is equivalent to rendering the tiger in very high resolution.

The stats improve steadily for each test configuration. The difference between traditional polygon approximation and cubic curve rendering with FP24 is shown in table 8.3.5.

Table 8.5: Measured improvements for Tiger Zoom. Polygonal vs. FP24.

Statistic	From	To	Improvement
Subdivision Overhead	887%	215%	75.7%
Triangle Count	36,256	18,084	50.0%
Vertex Count	18,252	8,752	52.0%
Tile List Command Count	35,526	16,290	54.1%

Memory traffic due to geometry and tile lists can be reduced by over 50%, and the rasterizable segment count is reduced by a factor of 75.7%. The subdivision overhead is 215% in the best configuration, which means that each segment is split more than one time in average.

The dividing triangle algorithm reduces tile lists by 16.3% for the polygonal approximation but only 5.3% for the best configuration.

Additional testing was performed to see how much the issue with concave control polygons affects the result. Under "Additional Tests" in appendix B, I have run two more test configurations for the Tiger Zoom case.

Cubic curve rendering with Great Precision give the best results, but they are not optimal. Table 8.3.5 lists the improvement of cubic curve rendering with FP23 over polygonal approximation when subdivision due to concave control polygons is deactivated.

Table 8.6: Measured improvements for Tiger Zoom. Polygonal vs. FP24 with concave CP support.

Statistic	From	To	Improvement
Subdivision Overhead	887%	114%	87.1%
Triangle Count	36,256	11,864	67.3%
Vertex Count	18,252	5,135	71.9%
Tile List Command Count	35,526	11,030	69.0%

When subdivision due to concave control polygons is deactivated, subdivision overhead is at 114%. This means that very little subdivision is performed, and the algorithm thus runs in almost linear time on the CPU. If concave control polygons can be supported, memory traffic due to geometry and tile lists can be reduced with around 70% over polygonal approximation for this test case.

8.4 Discussion

Of all the cases, The Chinese Text shows least improvement from the new approach. It has many small segments and most are only slightly curved. The traditional polygonal approximation technique succeeds in approximating almost all segments with lines without subdividing them. The Tiger test case also has many small segments, but they are more curved than in the Chinese Text case. It shows the second worst improvement from the new approach. This shows that paths with many small segments gain little improvement from the new approach.

The Zoomed Tiger shows very good improvement with the new method. With FP24 and support for curves with concave control polygons, little subdivision needs to be performed. In this case, memory

traffic due to both geometry and tile lists can be reduced by about 70%, and subdivision overhead is reduced by over 75%. FP16 seems to be a bit limiting compared to FP24, resulting in some more subdivision. Fixed-function rasterization needs to convert to quadratic curves, which results in even more subdivision. These more limited configurations still show a significant improvement over polygonal approximation. The Cubic Dude shows similar results. The Zoomed Tiger and the Cubic Dude show that paths with large cubic segments gain great improvement from the new approaches, especially on programmable GPUs with at least FP24 precision.

An internal precision of FP24 appears to be sufficient for cubic curve rasterization. *Great precision* shows neglible reduction in subdivision overhead compared to FP24 for both the Cubic Dude and the Zoomed Tiger.

The best improvement is shown for the Quadratic Guy. While polygonal approximation has a subdivision overhead of 540%, the new algorithm does not perform any subdivision at all, even on the most limited configurations. Memory traffic due to geometry and tile lists is reduced by over 65%. The new approach appears to be especially good at rasterizing quadratic curves with very little subdivision. The Quadratic Guy shows especially good improvement because he consists of large, curved segments that require much subdivision to generate a good polygonal approximation.

It appears that the fixed-function approach with a 256x256 texture is sufficient for rasterizing even quite large curves without subdivision. On programmable hardware, an internal precision of FP16 seems sufficient.

Vertex and polygon counts are slightly higher when using traditional triangle fan triangulation than when using the dividing triangle algorithm. This is because the triangle fan is drawn towards the centroid of the polygon, thus introducing a new vertex and one redundant triangle for each polygon.

The dividing triangle approach shows an improvement over traditional triangulation with a triangle fan towards the centroid. It is most useful to consider the configurations using polygonal approximations when measuring this improvement. Configurations using the new curve rasterization techniques include statistics from triangles that are not part of polygons rendered with the stencil algorithm, polluting the results. The best improvement due to the dividing triangle approach is shown for the Tiger Test, where the tile list command count decreases by 31%. The worst result is shown for the Chinese Text, where the tile list command count in fact increases by 4% with the dividing triangle approach. Average reduction in tile list commands may seem to lie somewhere around $20-25\%$.

It is apparent that the issue with concave control polygons limits performance. When subdivision due to this case is deactivated, the subdivision overhead of the decreases by 33.7% for the Cubic Dude, and by 47.1% for the Zoomed Tiger. It is clearly worth trying to solve the problem with direct rasterization of curves with concave control polygons.

The benchmarks indicate that the new approach can perform very efficient rasterization of quadratic curves on all configurations, and reasonably efficient rasterization of cubic curves, especially with FP24 precision. The issue with direct rasterization of concave control polygons should be fixed to gain the most improvement when rendering cubic curves. The dividing triangle approach seems to be around $20\% - 25\%$ more efficient than the triangle fan approach for rasterizing polygons with the stencil algorithm.

9 Conclusions

Recently developed algorithms promise significant improvement over traditional methods, but do not give guarantees about rasterization error. However, OpenVG has clearly defined criterions for how much error is allowed in rasterization of paths. Some features required for efficient OpenVG rasterization is also lacking.

The new approach by Loop and Blinn rasterize Bézier curves directly by drawing a simple bounding polygon around the segment and using the fragment shader to discard pixels that are not inside the curve [37]. Kokojima et al fill the remaining interior polygon using a method known as the stencil algorithm 2.10.4. I have found evidence of only one previous partial implementation of the techniques [46].

I have developed these algorithms further for use in an OpenVG implementation. Among other things, methods for efficient rendering of quadratic curves and elliptical arcs on fixed-function as well as programmable GPUs have been developed and precision is improved for quadratic curves. However, one of the main contributions of this thesis is the work on robustness that allows the approach to fulfil the requirements of the OpenVG specification. A new and more efficient approach is used for triangulation for the stencil algorithm.

My solution efficiently fills the interior of OpenVG paths. It supports both fixed-function and programmable GPUs. Paths may consist of lines, quadratic curves and cubic curves. (Cubic curves are only supported on programmable GPUs.) An approach for rendering elliptical arcs as well as methods for stroking and dashing is partially described but not implemented.

A prototype has been implemented on top of OpenGL. Preliminary verification and testing have shown that it works as expected and that the output looks correct. Further verification is required to determine whether the output is always in conformance with the OpenVG specification.

Benchmarking has been performed on realistic data-sets to measure the benefits of the new approach. Since it makes little sense to measure the rendering time of the prototype, implementation- and platform-independent statistics are collected. *Subdivision overhead* indicate scalability and load on the CPU as well as other parts of the system. *Vertex* and *triangle count* affects the amount of memory traffic due to transfer of geometry from the CPU to the GPU. The *tile list command count* affects the amount of memory traffic due to tile lists in a tile based renderer. It is also related to the cacheability of the render target in an immediate mode renderer, and thus the amount of memory traffic due to render target access.

The benchmarks show that vertex and triangle count as well as the simulated tile list count is lowered significantly in many common cases. Large reductions of up to 70% of geometry data are shown for big, smoothly curved shapes.

There is little improvement for detailed graphics such as Chinese text. The reason is that small details can be easily approximated without introducing much error. Big curves are however not easily approximated by lines, so that is where the new technique excels.

Subdivision on the CPU is usually avoided or greatly reduced, provided the GPU can rasterize segments with sufficient precision. Cubic curves must often be subdivided on GPUs that support only FP16, OpenGL ES' minimum precision requirement.

A programmable GPU with FP24 precision is required to gain the most benefit from the new approach

for cubic curve rasterization. Quadratic curve rasterization however can be done extremely efficiently on a simple fixed-function pipeline using a 256x256 look-up texture, or a programmable GPU with only FP16 precision.

The new triangulation method for the stencil algorithm performs in average around $20-25\%$ better than a simple triangle fan towards the centroid.

Unlike state of the art vector graphics solutions, the new approach performs only a minimum of processing in the CPU, involve much less memory traffic, and generally use the GPU in a more optimal way.

10 Future Work

10.1 Discussion of the Requirement Specification

The main task defined by the project assignment was to describe an algorithm for rasterization of filled OpenVG paths using a handheld GPU. In addition, it defines many optional tasks that can be performed if time permits. A requirement specification based on these tasks was presented in the interpretation of the assignment (chapter). Below is a description of which points of the requirement specification were completed and which were left for future work.

The first requirement states that the new approach and prototype should provide a significant improvement over traditional methods. The algorithms that have based my work on show an improvement over the traditional methods [37] [32] My own benchmarking also shows significant improvement with my approach. However, further benchmarking is desirable. See chapter 10.1.1.

Requirement number 2 states that efficient solutions for both fixed-function and programmable GPUs should be supported. The next requirement states that support for all paints and blend modes should be implementable at a later time. These requirements are fulfilled by my approach and implementation.

The fourth requirement states that algorithms must be robust and rasterization should be according to the OpenVG specification. A solution for elliptical arcs is only partially described and not implemented. My implementation is therefore not able to rasterize paths that include elliptical arcs, but these seem uncommon. This is left for future work as described in chapter 10.1.2

I have done my best to be sure that all special cases have been identified and taken care of. However, the approach explained in chapter 6.2.4 for approximating a cubic curve with a quadratic curve may not be robust. This must be taken care of in the future as described in chapter 10.1.3

Also, verification is required to make sure that my implementation is indeed according to the OpenVG specification. My verification is only partial, and further verification is required. It is thus left for future work to determine whether my solution successfully fulfils requirement 4, as described in chapter 10.1.4

The last requirement states that both stroking and filling of paths should be supported. An approach for filling paths with both fill rules is described and implemented. Stroking is however only partially described and not implemented. Half of requirement 5 is thus left for future work as described in chapter 10.1.5

10.1.1 Extensive Benchmarking

The prototype implementation should be compared against other software such as AmanithVG and Cairo. It is essential that the testing is performed with the same error thresholds. Extensive verification of the prototype should also be done to ensure that the statistics are representative.

10.1.2 Elliptical Arcs

Methods for approximating elliptical arcs with quadratic curves and lines are only partially described and not implemented. This should be done to fulfil the requirement specification.

10.1.3 Improve Cubic Curve Approximation Methods

The current method for approximating a cubic curve with a line (Cubic::ApproximateWithLine) is fairly conservative and should be replaced to avoid unnecessary subdivision. The method described in [23] seems promising as it is less conservative and still very cheap.

I am also not sure that the method for converting from cubic curve to quadratic curve (Cubic::ApproximateWithQuad is robust. A less conservative error estimation may also here be beneficial.

10.1.4 Extensive Verification

More extensive functionality testing should be performed. Verification of maximum rasterization error has not been performed. See chapter 7.4 for a description of how this can be done.

10.1.5 Stroking

The main task that remains in implementing stroking is offset curve generation. I have shown how Tiller and Hanson's approach [47] can be used to efficiently generate an offset curve to quadratic curves in chapter 6.2.6. Similar approaches must be developed for cubic curves and elliptical arcs. The same general approach can be used. As mentioned in 2.9, the topic of error estimation needs more research.

10.2 Additional Tasks

10.2.1 Running the Conformance Suite

As explained in chapter 7.1, an OpenVG conformance suite is available from the Khronos Group for testing whether implementations render according to the specification. It would be very interesting to run parts of the OpenVG conformance suite on the prototype to see whether the implementation actually conforms, and if not, to uncover remaining rasterization robustness issues.

10.2.2 Dashing

Dashing was not explicitly mentioned in the assignment text, and I interpreted it such that it was not a requirement.

To support dashing, the input path is split into many short pieces before generating the stroke path. This involves calculating arc lengths, which is not trivial and can be expensive for some segment types. It might be beneficial to first eliminate cubic segments by converting them to quadratic segments.

10.2.3 Implement a More Effective Subdivision Algorithm

The recursive subdivision algorithm is simple and relatively efficient. However, it does not give the most optimal results. Better results can be obtained by splitting segments at the point with the highest error instead of always splitting at the middle.

Other algorithms exist that produce fewer segments than the recursive subdivision algorithm. For example, the algorithm described in [27] is claimed to generate 70% of the vertices as methods based on recursive subdivision.

Better algorithms should be implemented both to improve performance for the new approach and to see how well it performs in comparison with a more efficient polygonal approximation.

10.2.4 Cheaper and More Accurate Estimation of Rasterization Error

An essential part of the robust OpenVG rasterizer using the new approach is the calculation of the rasterization error due to limited internal precision in the GPU.

The approach presented in chapter 5.2.5 is rather unscientific and involves a large number of operations. I have included some results that did not lead me to a complete solution for calculation of error, but may serve as a starting point for development of an alternative approach.

I will define three functions that represent the fragment shader expressions for quadratic curves, cubic curves and elliptical arcs. I call these functions $f_{quadratic}(u,v)$, $f_{cubic}(u,v,w)$ and $f_{elliptical}(u,v)$. When reasoning about a function I will treat the comparison as a special subtraction that does not introduce rounding error. (See chapter 5.2.5.)

Since varyings are linearly interpolated by the GPU, derivatives of u, v and w with respect to x and y are constant and can be easily calculated from 3 vertices of the boundary polygon. u, v and w can then be expressed as functions of x and y using these.

I will use the notation $max(t)$ and $min(t)$ for the highest and lowest values of a term t in the boundary polygon. Since varyings are linearly interpolated, their extreme values are always at one of the vertices and can thus easily be found. For more complex terms, pessimistic expressions that are cheap to evaluate can be found. Note that using these values, $max(|t|)$ and $min(|t|)$ can also be found trivially.

The following notation: $\bar{t}$ denotes the *absolute* error of t, while $\grave{t}$ denotes the *relative* error of t.

As explained in chapter 5.2.5, the relative rounding error bound $\grave{r}$ is equal to $2^{-mantissaBits-1}$.

Relative and Absolute Error

The terms absolute and relative error are essential to my estimation of rasterization error.

Please refer to an engineering mathematics book such as [33] for definition of the terms.

Pessimistic conversion from relative to absolute error bounds can be done by multiplying the relative error bound of t with $max(|t|)$.

Pessimistic conversion from absolute to relative error bounds can be done similarly by dividing the absolute error of t with $min(|t|)$.

Errors From Fragment Shader Expressions

Using the rules from chapter 10.2.4 and 5.2.5 I can formulate the error bounds for evaluation of f_{cubic}, $f_{quadratic}$ and $f_{elliptical}$ in a GPU.

Cubic Curves

Formula:
$f_{cubic}(u,v,w) = u^3 < vw$

u^3 can be written as $u*u*u$. Each instance of u has an implicit rounding error bound $\grave{r}$. In addition, each multiplication introduces an additional error bound $\grave{r}$ due to rounding. Multiplication adds relative error bounds. The result is a relative error bound $5\grave{r}$ for the expression.

Similarly, the relative error bound of vw is $3\grave{r}$.

Comparison adds the absolute errors of the operands. The values of $max(u^3)$ and $max(vw)$ are needed for conversion from relative to absolute error. $max(u^3)$ equals $max(u)^3$, in which $max(u)$ is easily found by considering only the vertices of the boundary polygon. The pessimistic estimate $m = max(|max(v)*max(w)|, |min(v)*min(w)|)$ can be easily found and is used in place of $max(vw)$ for our purpose.

Thus, an absolute error estimate of the whole expression is given by:
$\bar{f}_{cubic}(u,v,w) = 5\grave{r}max(u)^3 + 3\grave{r}m$

Quadratic Curves

Formula:
$f_{quadratic} = u^2 < v$

u^2 can be written as $u*u$. Each instance of u has an implicit error bound $\grave{r}$, and an additional $\grave{r}$ is introduced by the multiplication due to rounding. The result is a relative error bound of $3\grave{r}$ for the expression.

The implicit relative error of v is $\grave{r}$.

The comparison adds the absolute errors of the operands. The values of $max(u^2)$ and $max(v)$ are needed for conversion from relative to absolute error. $max(u^2)$ is equal to $max(|u|)^2$. Both $max(|u|)$ and $max(v)$ are easily found by considering only the vertices of the boundary polygon.

Thus, an absolute error estimate of the whole expression is given by:
$\bar{f}_{quadratic} = 3\grave{r}max(|u|)^2 + \grave{r}max(v)$.

Quadratic curves are drawn with a fixed set of varyings where u and v both vary between -1 and 1. Inserting the known values, the expression simplifies to a constant:
$\bar{f}_{quadratic} = 3\grave{r} * 1.0^2 + \grave{r} * 1.0 = 5\grave{r}$

Elliptical Arc

Formula:
$f_{elliptical}(u,v) = u^2 + v^2 < 1.0$

u^2 can be written as $u * u$. Each instance of u has an implicit rounding error bound $\grave{r}$. In addition, the multiplication introduces an additional error $\grave{r}$ due to rounding. This results in a relative error of $2\grave{r} + \grave{r} = 3\grave{r}$.

Conversion to absolute error requires $max(u^2)$, which is equal to $max(|u|)^2$. The absolute error is thus $3\grave{r} * max(|u|)^2$.

Equivalently, the absolute error of v^2 is $3\grave{r} * max(|v|)^2$.

Comparison is against 1.0, which can be represented exactly. The result therefore has the same absolute error as the first operand.

Thus, an absolute error estimate of the whole expression is given by:
$\bar{f}_{elliptical} = 3\grave{r} * max(|u|)^2 + 3\grave{r} * max(|v|)^2$

The unit circle is contained within $-1 < u < 1$ and $-1 < v < 1$. The bounding polygon can easily be constructed so that the varyings are always within this range. Inserting the known extreme values for u and v, the expression simplifies to a constant:
$\bar{f}_{elliptical} = 3\grave{r} * 1.0^2 + 3\grave{r} * 1.0^2 = 6\grave{r}$

Converting Absolute Error To Pixel Units

I have now presented an approach for calculating the maximum absolute error of the fragment shader expression within the boundary polygon. This error must now be transformed into surface space so that its magnitude can be compared against maxDistance.

If the distance between the real curve and the rasterized curve is measured along the real curve's normal, this gives a pessimistic estimate, which is fine for our purposes. The curve normal is given by the *gradient* of the implicit function.

Unfortunately, this was as far as I got with this approach because of lack of time. The approach described in chapter 5.2.5 is however good enough for our purposes.

10.2.5 Concave Control Polygons

Work should be done on direct rendering of cubic curves. (The current solution is to subdivide until the problem goes away.) The benchmarks show that there is much to gain if this problem can be solved.

Something can probably be done so that they can be rasterized directly by evaluating the implicit equation in the fragment shader. This will avoid subdivision in these cases and reduce the triangle and vertex count.

10.2.6 Path Simplification

Detailed paths with many small segments such as tiger.path, gain little from the new approach. This is because small segments are easily approximated with simple lines, and no further simplification is possible.

In these cases it should however be possible to approximate multiple segments with a single segment to simplify the geometry.

A possible approach for generating the approximation is try to fit a cubic curve to the vertices of the interior polygon. If the cubic curve goes through the vertices of the interior polygon, it will typically follow the intended outline of the shape. (Elliptical arcs and quadratic segments can also be used and may be easier/cheaper to apply.)

A conservative estimation of the maximum distance from the approximation to the actual curve must then be calculated. The error must be compared to maxDistance together with rasterization and maxSnapError to determine whether the approximation is good enough that it can be used.

Path simplification using cubic curves should increase performance for scaled down and detailed paths, and also gain the same benefit from the new approach as simple paths.

10.2.7 Caching of Approximated Paths

A common use case is for an application to render the same path multiple times with different transformation matrix over one or more frames. This opens for optimization by caching of geometry. Subdivisions and calculations of varyings can be done once and used in multiple different transformation matrices as long as the scale is the same. (When scale changes, error comparisons will give different results and lead to different subdivision.) Multiple subdivided versions for different scale factors can also be cached.

This can be combined with the approach from chapter 10.2.6 to support dynamic level-of-detail through cached geometry.

10.2.8 Anti-aliasing

I have not yet mentioned anti-aliasing. This term refers to techniques intended to reduce the jagged appearance of diagonal lines and silhouettes. (This jagged appearance can be seen in figure 2.6.)

Modern GPUs have support for anti-aliasing through multisampling and supersampling. In the case of rendering curves by evaluating implicit equations in the fragment shader, supersampling must be used so that the fragment shader is executed once for each fragment. Supersampling means to perform rasterization at a high resolution and then scaling down with filtering, and is thus a slow brute force approach to anti-aliasing. By computing the coverage in the fragment shader much faster anti-aliasing is possible with comparable quality.

Both Loop and Blinn's paper and Kokojima et al's sketch present methods for anti-aliasing by estimating the coverage of the fragment. This area should be researched because it to see if it can be applied.

10.2.9 Evaluation of Visual Quality With Different Techniques

OpenVG specifies two rendering profiles: FASTER and BETTER. Curves rasterized with polygonal approximation using maxDistance= 1.0 do not look completely smooth. When rendering with the FASTER profile, the value of maxDistance should be as high as possible for performance reasons. However, when rendering with the BETTER profile, the value should be low enough that the curve appears completely smooth under most conditions.

Different rasterization methods have different visual qualities, and may tolerate different values for maxDistance while still looking smooth. The traditional polygonal approximation approach will create outlines that are slightly shorter than the path they represent and appear coarse. I expect that the approach of evaluating implicit equations in the fragment shader will create a pixelated outline, and will only show artifacts when rendering large curve segments.

It is possible that the new approaches have a visual advantage over the traditional method. If that is the case, they can use a higher value for maxDistance than polygonal approximation when rendering with the BETTER profile and thus perform even faster. For example, it is expected that approximating cubic curves with quadratic curves will look better than approximating them with lines, since the curve remains

smooth. It is left for future work to see whether this is the case and to determine appropriate values for maxDistance when rendering with the BETTER profile.

The settings namespace in the prototype has two variables that can be used to affect the priority between approximation by lines and quadratic curves: lineApproximationScale and quadraticApproximationScale. These were added in the last minute and are therefore not described in chapter 6.3.4. Please refer to the source code reference manual in appendix C for more information.

10.2.10 Hardware Support For Curved Primitives

Industry-standard GPUs operate on triangles. It should be possible to implement support for additional primitives such as Bézier curves in the rasterizer.

One possibility is discussed in chapter 5.3. Another possibility is presented below.

Estimating the size of the required hardware for both techniques is left for future work.

Tessellation in Hardware

Some earlier GPUs for desktop systems had hardware support for tessellation of surface patches, three-dimensional equivalents of the quadratic and cubic curves [3]. A similar, but simpler approach would be possible for 2d vector graphics. Referring to the conceptual pipeline from chapter 2.4.2, the tessellation unit would sit between the primitive assembly and rasterizer stages. It would take a quadratic or cubic curve as input (or even an elliptical arc) and subdivide it into triangles until the maximum distance between the approximated geometry and the input curve was less than an application-specified threshold. Multipliers would be needed for the calculation of maximum error in the approximation, which is rather expensive in terms of die area.

In tile based renderers the benefits of this approach are limited. The tessellation process would have to be performed either before tiling or after the tile list command has been read, before rasterization. If it happens before tiling, the tile lists would still suffer from long, thin, diagonal triangles, as they do with CPU-based tessellation. If it happens before rasterization, the tessellation would have to be performed once for each tile covered by the segment.

Note that this idea is unrelated to most of the techniques presented in this thesis, as it represents an alternative approach to curve rasterization.

Bibliography

[1] Glitz (Home Page). http://www.freedesktop.org/wiki/Software/glitz.

[2] PowerVR White Paper. http://www.beyond3d.com/reviews/videologic/vivid/PowerVR_WhitePaper.pdf, november 2000.

[3] Truform White Paper. http://ati.amd.com/products/pdf/truform.pdf, 2001.

[4] NVIDIA GPU Programming Guide. http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf, 2005.

[5] OpenGL ES Common Profile Specification 2.0. http://www.khronos.org/cgi-bin/fetch/fetch.cgi?opengles_spec_2_0, 2005.

[6] OpenVG Specification 1.0.1. http://www.khronos.org/files/openvg_1_0_1.pdf, 2005.

[7] Gameboy. http://en.wikipedia.org/wiki/Gameboy, accessed 11th June 2007, December 2006.

[8] Nintendo DS. http://en.wikipedia.org/wiki/Nintendo_ds, accessed 11th June 2007, November 2006.

[9] NVIDIA Corporation. http://www.nvidia.com/, accessed 11th June 2007, 2006.

[10] Qt Benchmark. http://zrusin.blogspot.com/2006/10/benchmarks.html, accessed 11th June 2007, August 2006.

[11] The Khronos Group. http://www.khronos.org/, 2006.

[12] The OpenGL ES Shading Language. http://www.khronos.org/files/opengles_shading_language.pdf, 2006.

[13] Amanith Framework Performance (Product Home Page). http://www.amanithvg.com/performance.html, accessed 11th June 2007, june 2007.

[14] Graphics processing unit. http://en.wikipedia.org/wiki/GPU, accessed 11th June 2007, June 2007.

[15] Khronos OpenGL ES API Registry. http://www.khronos.org/registry/gles/, 2007.

[16] Qt by Trolltech (Product Home Page). http://trolltech.com/products/qt, accessed 11th June 2007, june 2007.

[17] The cairo graphics library (Home Page). http://cairographics.org/, 2007.

[18] AKENINE-MÖLLER, T., AND HAINES, E. Graphics Hardware. In *Real-Time Rendering*. A K Peters, 2002, ch. 15.

[19] ANTOCHI, I., JUURLINK, B., VASSILIADIS, S., AND LIUHA, P. Scene Management Models and Overlap Tests for Tile-Based Rendering. *Proceedings of the EUROMICRO Systems on Digital System Design (DSD04)* (2004).

[20] AUSTAD, T. Personal communication.
Employee, ARM Media Division.

[21] COMBA, J. L. D., DIETRICH, C. A., PAGOT, C. A., AND SCHEIDEGGER, C. E. Computation on GPUs: From a Programmable Pipeline to an Efficient Stream Processor. *RITA 10* (2003), 41–70.

[22] ELBER, G., LEE, I.-K., AND KIM, M.-S. Comparing offset curve approximation methods. *Computer Graphics and Applications, IEEE 17* (June 1997), 62–71.

[23] FISCHER, K. Piecewise linear approximation of bezier curves. `http://people.inf.ethz.ch/fischerk/pubs/bez.pdf`, October 2000.

[24] FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. *Computer Graphics Principles and Practice*. Addison-Wesley, 1996.

[25] FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. *Computer Graphics Principles and Practice*. Addison-Wesley, 1996, pp. 513–514.

[26] GROLEAU, T. Approximating Cubic Bezier Curves in Flash MX. `http://timotheegroleau.com/Flash/articles/cubic_bezier_in_flash.htm`, 2002.

[27] HAIN, T. F., AHMAD, A. L., RACHERLA, S. V. R., AND LANGAN, D. D. Fast, Precise Flattening of Cubic Bézier Path and Offset Curves. In *17th Brazilian Symposium on Computer Graphics and Image Processing* (October 2004), pp. 244–249.

[28] HOSCHEK, J. Spline Approximation of Offset Curves. *Computer Aided Graphics Design 5* (June 1988), 33–40.

[29] HOSCHEK, J., AND WISSEL, N. Optimal Approximate Conversion of Spline Curves and Spline Approximation of Offset Curves. *Computer-Aided Design 20* (October 1988), 475–483.

[30] JIM RUPPERT. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. *Journal of Algorithms, NASA Ames Research Center* (1995).

[31] KILGARIFF, E., AND FERNANDO, R. The GeForce 6 Series GPU Architecture. In *GPU Gems 2*, M. Pharr and R. Fernando, Eds. Addison-Wesley Professional, mar 2005, ch. 30.

[32] KOKOJIMA, Y., SUGITA, K., SAITO, T., AND TAKEMOTO, T. Resolution Independent Rendering of Deformable Vector Objects using Graphics Hardware. In *ACM SIGGRAPH 2006 Sketches*. ACM Press, 2006.

[33] KREYSZIG, E. *Advanced Engineering Mathematics*, 8 ed. John Wiley and Sons, Inc., 1999, ch. 17.

[34] LI, X.-Y. Spacing Control and Sliver-free Delaunay Mesh. In *Proc. 9th Int. Meshing Roundtable* (October 2000), Sandia Nat. Lab., pp. 295–306.

[35] LILAND, E., AND FIELDING, E. Multicore GPU Simulation, 2006.

[36] LILAND, E. L. Analysis of Geometry in Doom 3 (ARM Confidential). 2006.

[37] LOOP, C., AND BLINN, J. Resolution Independent Curve Rendering using Programmable Graphics Hardware. In *Proceedings of ACM SIGGRAPH* (july 2005), vol. 24, ACM Press, pp. 1000–1009.

[38] MAHONEY, J. M. 3D Graphics Then and Now: From the CPU to the GPU. In *Proceedings of the 5th Winona Computer Science Undergraduate Research Seminar* (apr 2005).

[39] MAISONOBE, L. Drawing an elliptical arc using polylines, quadratic or cubic Bézier curves. http://www.spaceroots.org/documents/ellipse/elliptical-arc.pdf, July 2003.

[40] ÅMODT, E. Personal communication.
Employee, ARM Media Division.

[41] PEDDIE, J. Graphics in handhelds. In *Handheld Multimedia Devices*. 2004, p. 99.

[42] PEDDIE, J. The need for Open Standards in the Embedded Market. http://www.khronos.org/developers/library/jpr_keynote.ppt, 2005.

[43] SEGAL, M., AND AKELEY, K. The OpenGL Graphics System: A Specification (Version 2.0). http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf, 2004.

[44] SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. *Drawing Filled, Concave Polygons Using the Stencil Buffer*, fourth ed. Addison-Wesley, 2004, ch. 14, pp. 600–601.

[45] SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. *OpenGL Programming Guide*, fourth ed. Addison-Wesley, 2004.

[46] STEFAN GUSTAVSON. Direct rendering of cubic Bezier contours in RSL. http://staffwww.itn.liu.se/~stegu/aqsis/implicitBeziers.pdf.

[47] TILLER, W., AND HANSON, E. G. Offsets of Two-Dimensional Profiles. In *Computer Graphics and Applications, IEEE* (1984), vol. 4, IEEE Computer Society, pp. 36–46.

A Prototype User Manual

This appendix provides a user manual for the prototype application.

A.1 Getting Started

Start the program without command-line arguments to view a set of auto-generated synthetic tests. Each test consists of a single path, and only one is shown at a time. The user can press p or right-click and select "Change Path" to cycle through the different tests.

Start the program with a filename as command-line argument to load a test from an external file. The file must be in the *path*-format used internally for testing by the OpenVG group at ARM Norway. A set of tests is supplied digitally with the thesis. There is one test in each file, but each one can consist of multiple paths with different colors, shown simultaneously. When editing geometry or viewing in wireframe, the user can press the p key or right-click and select "Change Path" to select which path to edit or view.

A.2 User Interface Overview

The application will launch two windows: The main window (figure A.1), and the console (A.2). The main window is used for displaying graphics and for user interaction. The console window prints statistics for benchmarking. This includes rasterizable segment count, triangle count, vertex count and tile list command count. The tile list command count is calculated with the assumption that the GPU is a tile based renderer with bounding box tiling and tiles with a dimension of 16x16 pixels. (The dimensions can be changed in settings.cpp)

Use the arrow keys to navigate, and the $+/-$ keys to zoom in and out.

Modifying geometry is done by dragging control points with the mouse. If the control points are not visible, press capital P (see chapter A.3.) If the displayed image consists of multiple paths, press the p key to change the active path. Use wireframe mode if you want to view only the path that is being modified.

To change rendering resolution, just stretch the window to the desired dimensions. The current resolution is shown in the top left corner.

Some options, mostly related to rendering mode, can be reached through a pop-up menu or keyboard shortcuts. The menu can be reached by right-clicking the main window.

A.3 Menu Choices and Keyboard Shortcuts

Many aspects of the rendering can be modified using either keyboard shortcuts or by right-clicking and selecting an option from the menu. The keyboard shortcuts are case-sensitive. The current rendering

Figure A.1: The main window of the prototype application.

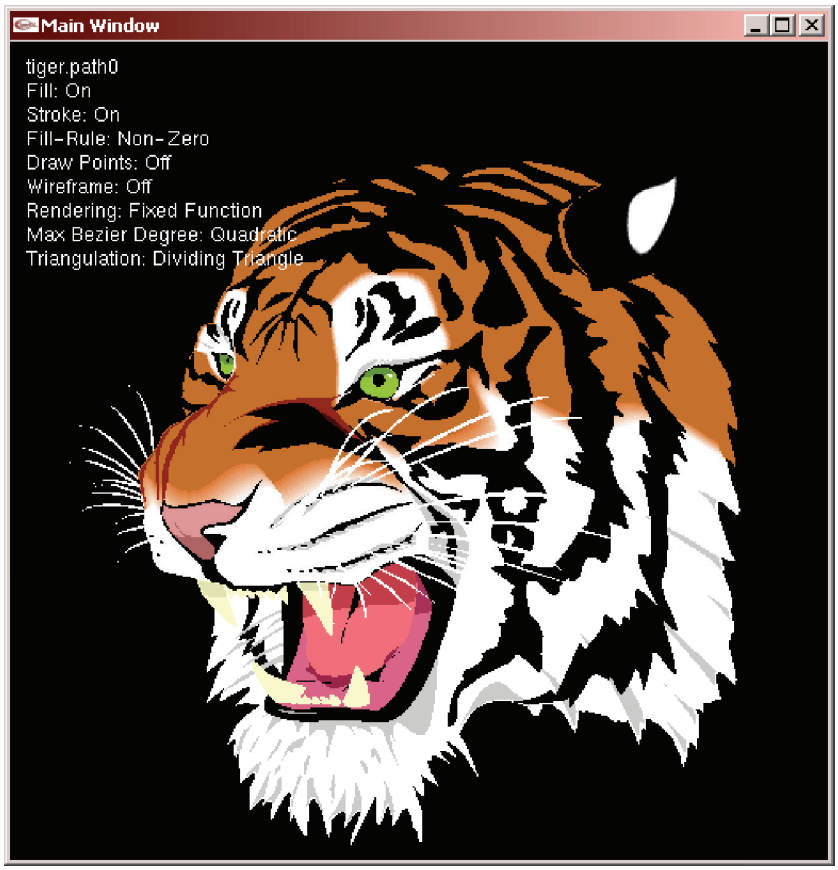
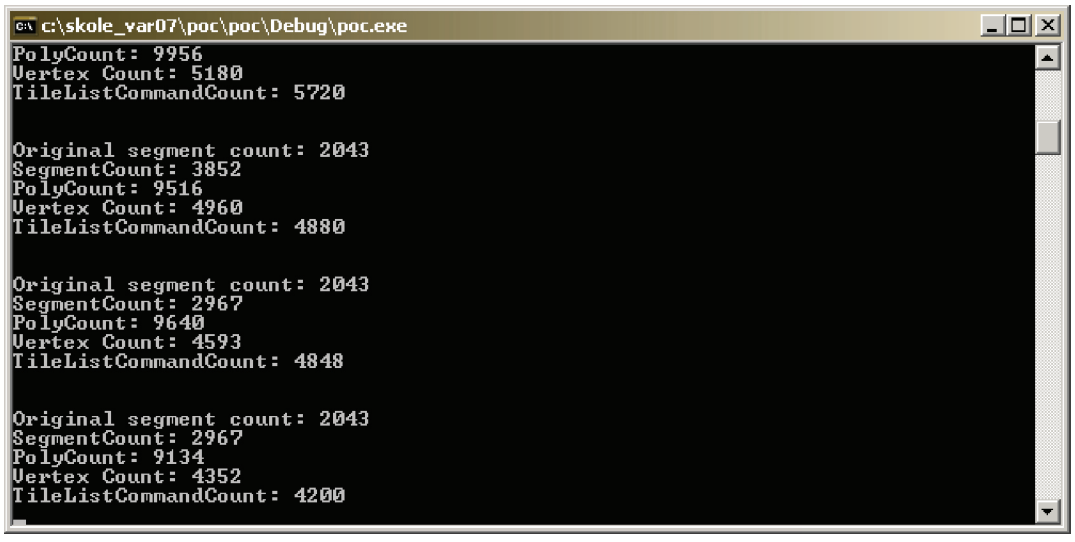


Figure A.2: The console window of the prototype application.



mode settings are shown in the top-left corner of the screen.

p - Change Path
Changes active path. If the program was run without command-line argument, this means that another synthetic test is shown. If the program was run with the name of a path-file in the command-line argument, this means that another path can be edited. Only the active path is shown when in wireframe mode.

f - Toggle Fill
Toggles filling of path. If you deactivate this, the selected path is not filled. Paths can be turned on and off individually in multi-path data-sets.

s - Toggle Stroke
Toggles stroking of path. Since stroking is not implemented, this option does nothing.

F - Next Fill Rule
Changes fill rule of selected path. Choose between odd/even and non-zero. The difference is apparent only if the path overlaps itself with the same orientation twice.

P - Draw Control Points
Activates or deactivates rendering of control points. Activate when you wish to manipulate geometry.

w - Toggle Wireframe
Toggle wireframe rendering. If viewing a multi-path data-set, only the selected path is drawn in this mode. This mode is thus also used to see which path is selected. (The selected path name is also printed at the top left of the window.)

r - Select Fixed-Function or Programmable Rendering
Changes between rendering with fixed-function and programmable pipeline functionality. The fixed-function rendering program path is such that it can be implemented on OpenGL ES 1.x hardware, while the programmable program path can be implemented on OpenGL ES 2.0.

If the programmable pipeline mode is activated on a device that does not support it, the image will look incorrect. It can still be desirable to activate this option to be able to extract statistics.

d - Maximum Bézier Degree
Changes maximum degree of Bézier curves that can be rasterized. First degree means that only lines can be rasterized, so the algorithm will convert the whole path to a polygon before rasterization. This is the traditional approach to path rasterization, and will be useful during benchmarking. Second degree means that lines and quadratic curves are allowed. Cubic curves will not be rasterized directly, but converted to quadratic curves. Third degree means that all curve types are allowed, including cubic curves. Note that cubic curves can not be rasterized directly in fixed-function mode.

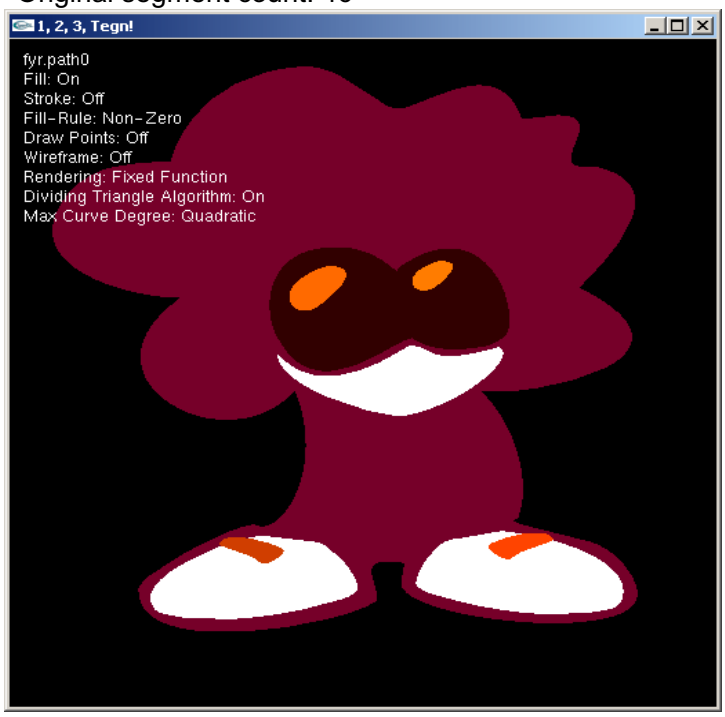
t - Toggle Triangulation Method
Changes maximum degree of Bézier curves that can be rasterized. First degree means that only lines can be rasterized, so the algorithm will convert the whole path to a polygon before rasterization. This is the traditional approach to path rasterization, and will be useful during benchmarking. Second degree means that lines and quadratic curves are allowed. Cubic curves will not be rasterized directly, but converted to quadratic curves. Third degree means that all curve types are allowed, including cubic curves. Note that cubic curves can not be rasterized directly in fixed-function mode.

B

Benchmark Results

Benchmark case 1: Cubic dude

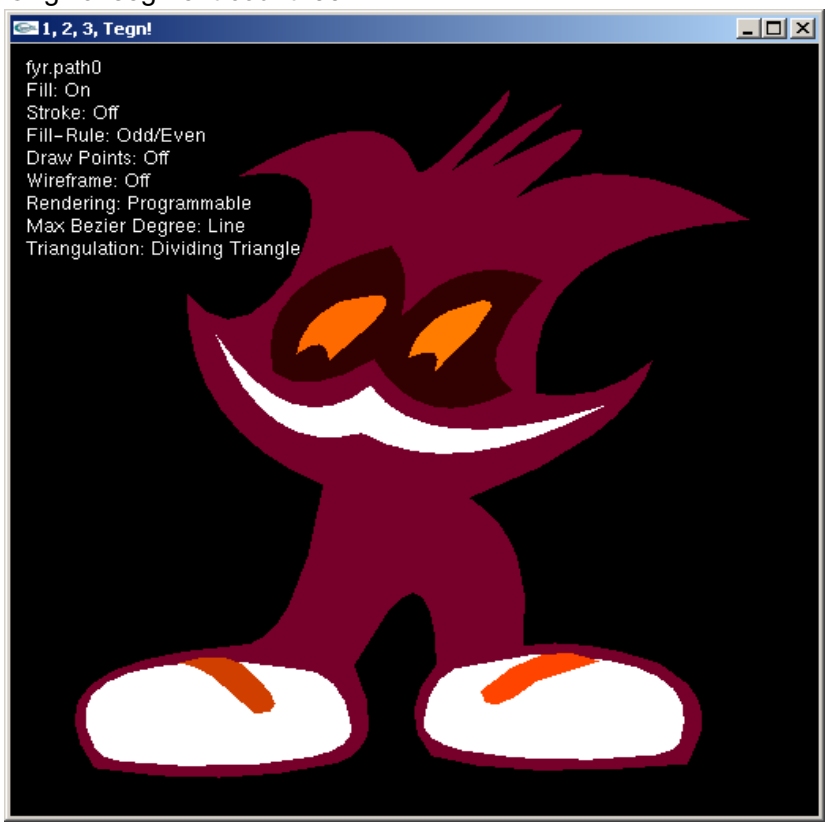
Original segment count: 46



	Triangle Fan	Dividing Triangle
Polygonal approximation	SegmentCount: 336 TriangleCount: 676 VertexCount: 347 TileListCommandCount: 564	SegmentCount: 336 TriangleCount: 654 VertexCount: 338 TileListCommandCount: 428
Quadratic Fixed-function 256x256	SegmentCount: 180 TriangleCount: 588 VertexCount: 303 TileListCommandCount: 428	SegmentCount: 180 TriangleCount: 556 VertexCount: 294 TileListCommandCount: 402
Quadratic Programmable FP16	SegmentCount: 175 TriangleCount: 574 VertexCount: 296 TileListCommandCount: 420	SegmentCount: 175 TriangleCount: 552 VertexCount: 287 TileListCommandCount: 390
Cubic Programmable FP16	SegmentCount: 99 TriangleCount: 448 VertexCount: 233 TileListCommandCount: 364	SegmentCount: 99 TriangleCount: 426 VertexCount: 224 TileListCommandCount: 340
Cubic Programmable FP24	SegmentCount: 86 TriangleCount: 402 VertexCount: 210 TileListCommandCount: 322	SegmentCount: 86 TriangleCount: 380 VertexCount: 201 TileListCommandCount: 302
Cubic Programmable Great Precision	Same as above	Same as above

Benchmark case 2: Quadratic Guy

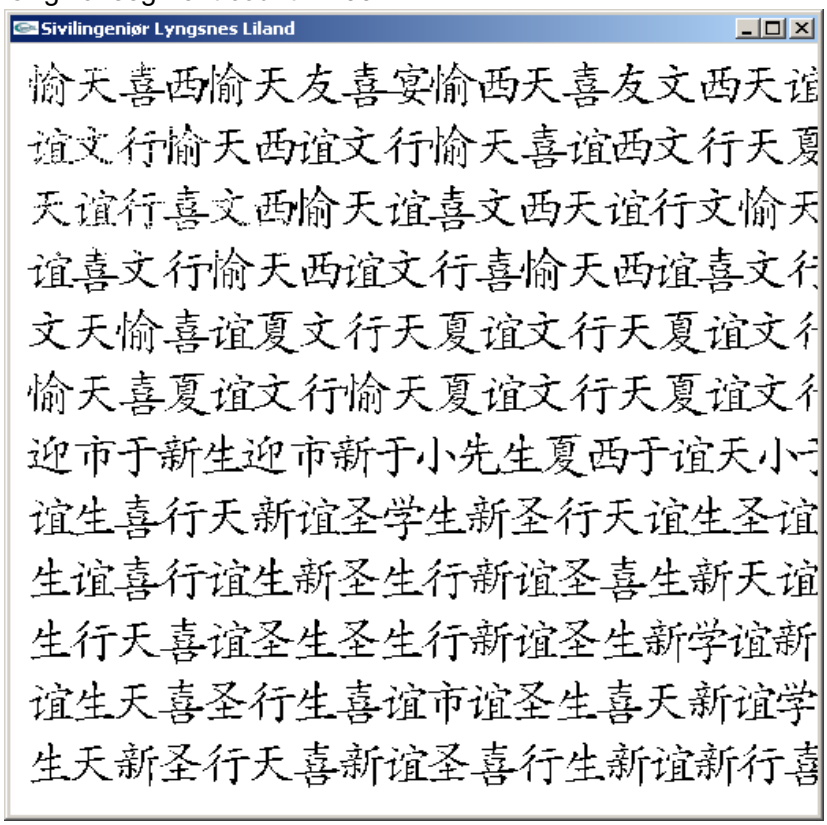
Original segment count: 53



	Triangle Fan	Dividing Triangle
Polygonal approximation	SegmentCount: 287 TriangleCount: 578 VertexCount: 298 TileListCommandCount: 530	SegmentCount: 287 TriangleCount: 556 VertexCount: 289 TileListCommandCount: 418
Quadratic Fixed-function 256x256	SegmentCount: 53 TriangleCount: 212 VertexCount: 115 TileListCommandCount: 202	SegmentCount: 53 TriangleCount: 190 VertexCount: 106 TileListCommandCount: 184
Quadratic Programmable FP16	Same as above	Same as above
Cubic Programmable FP16	Same as above	Same as above
Cubic Programmable Great Precision	Same as above	Same as above

Benchmark case 3: Chinese Text

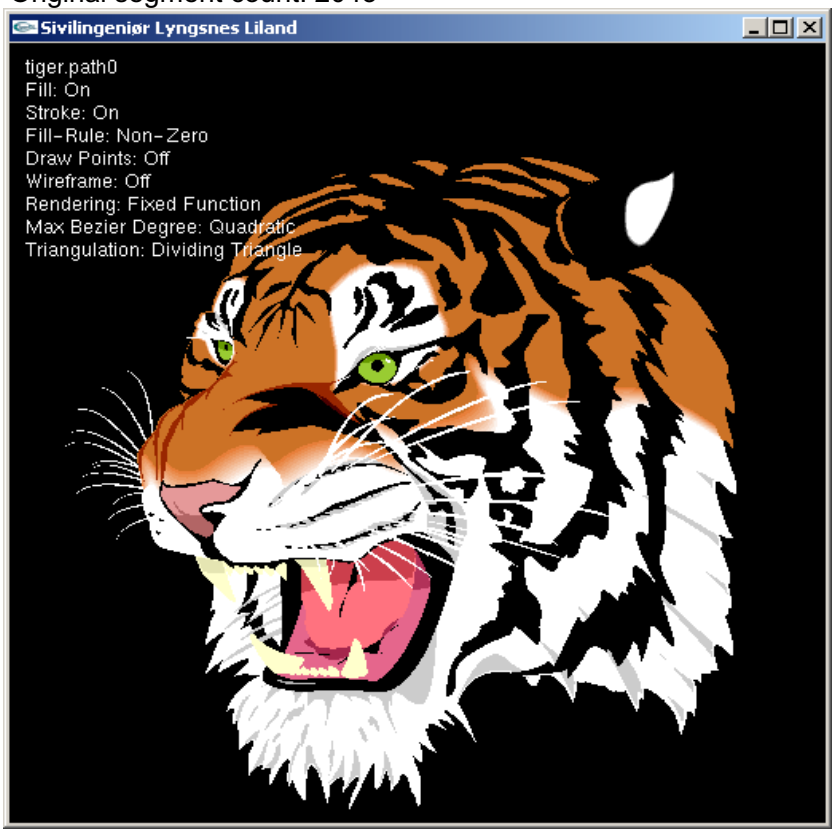
Original segment count: 27332



	Triangle Fan	Dividing Triangle
Polygonal approximation	SegmentCount: 27717 TriangleCount: 55434 VertexCount: 28068 TileListCommandCount: 5888	SegmentCount: 27717 TriangleCount: 52526 VertexCount: 26614 TileListCommandCount: 6130
Quadratic Fixed-function 256x256	SegmentCount: 27332 TriangleCount: 55434 VertexCount: 28068 TileListCommandCount: 5942	SegmentCount: 27332 TriangleCount: 52526 VertexCount: 26614 TileListCommandCount: 6044
Quadratic Programmable FP16	Same as above	Same as above
Cubic Programmable FP16	Same as above	Same as above
Cubic Programmable Great Precision	Same as above	Same as above

Benchmark case 4: Tiger

Original segment count: 2043



	Triangle Fan	Dividing Triangle
Polygonal approximation	SegmentCount: 4965 TriangleCount: 9956 VertexCount: 5180 TileListCommandCount: 5720	SegmentCount: 4965 TriangleCount: 9452 VertexCount: 4939 TileListCommandCount: 3924
Quadratic Fixed-function 256x256	SegmentCount: 3895 TriangleCount: 9560 VertexCount: 4982 TileListCommandCount: 4924	SegmentCount: 3895 TriangleCount: 9056 VertexCount: 4741 TileListCommandCount: 3980
Quadratic Programmable FP16	SegmentCount: 3852 TriangleCount: 9516 VertexCount: 4960 TileListCommandCount: 4880	SegmentCount: 3852 TriangleCount: 9012 VertexCount: 4719 TileListCommandCount: 3952
Cubic Programmable FP16	SegmentCount: 2967 TriangleCount: 9640 VertexCount: 4593 TileListCommandCount: 4848	SegmentCount: 2967 TriangleCount: 9134 VertexCount: 4352 TileListCommandCount: 4200
Cubic Programmable Great Precision	SegmentCount: 2966 TriangleCount: 9630 VertexCount: 4588 TileListCommandCount: 4838	SegmentCount: 2966 TriangleCount: 9124 VertexCount: 4347 TileListCommandCount: 4196

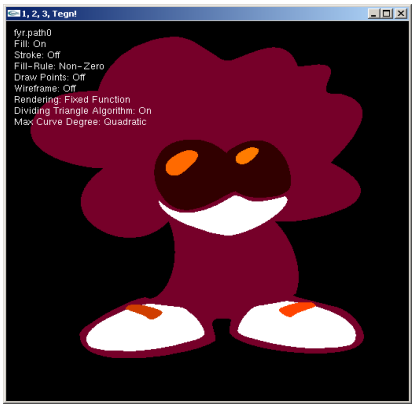
Benchmark case 5: Tiger (zoomed)

Original segment count: 2043

	Triangle Fan	Dividing Triangle
Polygonal approximation	SegmentCount: 18115 TriangleCount: 36256 VertexCount: 18252 TileListCommandCount: 35526	SegmentCount: 18115 TriangleCount: 35752 VertexCount: 18011 TileListCommandCount: 29752
Quadratic Fixed-function 256x256	SegmentCount: 13183 TriangleCount: 34010 VertexCount: 17131 TileListCommandCount: 31266	SegmentCount: 13183 TriangleCount: 33506 VertexCount: 16890 TileListCommandCount: 28702
Quadratic Programmable FP16	SegmentCount: 12734 TriangleCount: 33450 VertexCount: 16851 TileListCommandCount: 30810	SegmentCount: 12734 TriangleCount: 32946 VertexCount: 16610 TileListCommandCount: 28332
Cubic Programmable FP16	SegmentCount: 5387 TriangleCount: 21412 VertexCount: 10415 TileListCommandCount: 19832	SegmentCount: 5387 TriangleCount: 20906 VertexCount: 10174 TileListCommandCount: 18898
Cubic Programmable FP24	SegmentCount: 4393 TriangleCount: 18590 VertexCount: 8993 TileListCommandCount: 17204	SegmentCount: 4393 TriangleCount: 18084 VertexCount: 8752 TileListCommandCount: 16290
Cubic Programmable Great Precision	SegmentCount: 4378 TriangleCount: 18560 VertexCount: 8978 TileListCommandCount: 17188	SegmentCount: 4378 TriangleCount: 18054 VertexCount: 8737 TileListCommandCount: 16270

Additional Tests

Case 6: Concave Control Triangles Allowed



	Cubic Dude	**Tiger Zoomed**
Cubic Programmable FP24 Concave CP Dividing Triangle	SegmentCount: 57 TriangleCount: 294 VertexCount: 158 TileListCommandCount: 272	SegmentCount: 2323 TriangleCount: 11864 VertexCount: 5135 TileListCommandCount: 11030
Cubic Programmable Great Precision Concave CP Dividing Triangle	Same as above	SegmentCount: 2304 TriangleCount: 11830 VertexCount: 5118 TileListCommandCount: 11008

C

Source Code Reference Manual

Prototype Reference Manual

Generated by Doxygen 1.3.9.1

Mon Jun 11 23:57:25 2007

Contents

Chapter 1

Prototype Namespace Index

1.1 Prototype Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 2

Prototype Hierarchical Index

2.1 Prototype Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Prototype Class Index

3.1 Prototype Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

Prototype Namespace Documentation

4.1 settings Namespace Reference

Variables

- bool **usePrammablePipeline** = true
- float **maxError** = 1.0
- float **maxSnapError** = sqrt(1.0/16 * 1.0/16 + 1.0/16 * 1.0/16) * 0.5
- int **maxDegree** = 3
- int **pixelShaderMantissaBits** = 10
- const int **quadraticLutWidth** = 256
- const int **quadraticLutHeight** = 256
- int **tileDimension** = 16
- bool **useDividingTriangle** = true
- float **lineApproximationScale** = 1.f
- float **quadraticApproximationScale** = 1.f
- bool **convertCubicToQuadraticHack** = false

4.1.1 Detailed Description

This namespace specifies rendering options that are intended to be constant for a target device, but may be changed for benchmarking when using the prototype. Some of these can be changed from the application UI.

4.1.2 Variable Documentation

4.1.2.1 bool settings::convertCubicToQuadraticHack = false

Set to convert cubic curves to quadratic curves in the .path loader. No subdivision is performed. This is a hack used to generate a "realistic" testset based on quadratic curves. It hangs for some inputs.

4.1.2.2 float settings::lineApproximationScale = 1.f

Used to scale approximation error from approximation with line and quadratic. It may help visual quality to prefer quadratic approximation over linear.

4.1.2.3 int settings::maxDegree = 3

Specifies the maximum allowed degree of bezier curves. 1=line, 2=quadratic, 3=cubic. This can be used for comparing the results of a traditional polygonal approximation with direct cubic segment rasterization.

4.1.2.4 float settings::maxError = 1.0

Maximum rasterization error. Specifies the maximum difference between the real path and the rasterized result. Should be 1.0 or less to conform with the OpenVG specification.

4.1.2.5 float settings::maxSnapError = sqrt(1.0/16 * 1.0/16 + 1.0/16 * 1.0/16) * 0.5

Specifies the inherent error from the GPU's rasterizer. The default value is the error generated by snapping vertices to a fine grid with 16x16 nodes per pixel.

4.1.2.6 int settings::pixelShaderMantissaBits = 10

Specifies the internal precision in the target pixel shader unit. It is used for calculating rasterization error when using Loop and Blinn's algorithm for rasterizing quadratic and cubic curves. The default value is 10, which is the minimum precision required by GLESSL

4.1.2.7 float settings::quadraticApproximationScale = 1.f

Used to scale approximation error from approximation with line and quadratic. It may help visual quality to prefer quadratic approximation over linear.

4.1.2.8 const int settings::quadraticLutHeight = 256

Specifies the dimensions of the look-up texture used for rasterizing quadratic curves in the fixed-function pipeline approach.

4.1.2.9 const int settings::quadraticLutWidth = 256

Specifies the dimensions of the look-up texture used for rasterizing quadratic curves in the fixed-function pipeline approach.

4.1.2.10 int settings::tileDimension = 16

Used as basis for the calculation of tile list command count. The default value is 16, which specifies that the GPU uses tiles with 16x16 pixels.

4.1.2.11 bool settings::useDividingTriangle = true

Decides whether the dividing triangle or triangle fan triangulation approach is used for the stencil algorithm when rasterizing interior polygons

4.1.2.12 bool settings::useProgrammablePipeline = true

use the programmable pipeline approach? if false, use the fixed-function approach

4.2 shader_fw Namespace Reference

Functions

- bool **HaveShaders** ()
- void **PrintShaderInfoLog** (GLuint obj)
- void **PrintProgramInfoLog** (GLuint obj)

4.2.1 Detailed Description

Namespace for utility functions related to opengl shaders.

4.2.2 Function Documentation

4.2.2.1 bool shader_fw::HaveShaders ()

Checks whether the host device and drivers support the programmable GPU approach.

Returns:
Does the host support shaders

4.2.2.2 void shader_fw::PrintProgramInfoLog (GLuint *obj*)

Prints the OpenGL program info log to the console

4.2.2.3 void shader_fw::PrintShaderInfoLog (GLuint *obj*)

Prints the OpenGL shader info log to the console

4.3 stats Namespace Reference

Functions

- void **NewTriangle** (**Vector2** p0, **Vector2** p1, **Vector2** p2)
- void **NewFrame** ()
- void **NewSegment** (int op)
- void **PrintFrameStats** ()

Variables

- **Stats frame**

4.3.1 Detailed Description

This namespace contains tools for collecting data for a single frame.

4.3.2 Function Documentation

4.3.2.1 void stats::NewFrame ()

Reset the counters in stats::frame. Should be called before starting a new frame.

4.3.2.2 void stats::NewSegment (int *op* = 1)

Call this to add one or more segments to the segment count

Parameters:
op Number of segments to add

4.3.2.3 void stats::NewTriangle (Vector2 *p0*, Vector2 *p1*, Vector2 *p2*)

Call this to collect statistics for a triangle

Parameters:
p0 Triangle vertex
p1 Triangle vertex
p2 Triangle vertex

4.3.2.4 void stats::PrintFrameStats ()

Print collected statistics for current frame.

Chapter 5

Prototype Class Documentation

5.1 BBox2 Struct Reference

`#include <vecmath.h>`

Public Member Functions

- void **contain** (**Vector2** op)
- **Vector2 dimensions** ()
- float **area** ()

Public Attributes

- bool **inited**
- **Vector2 min**
- **Vector2 max**

5.1.1 Detailed Description

Two-dimensional bounding box class

5.1.2 Member Function Documentation

5.1.2.1 float BBox2::area () `[inline]`

Calculate the area of the bounding box

Returns:
The area of the bounding box

5.1.2.2 void BBox2::contain (Vector2 *op*) `[inline]`

If specified point is outside bounding box, extend the box so that it is inside.

Parameters:
op A point to include in the bounding box

5.1.2.3 Vector2 BBox2::dimensions () [inline]

Calculate the dimensions of the bounding box

Returns:
Dimensions of the bbox

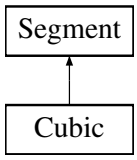
The documentation for this struct was generated from the following file:

- vecmath.h

5.2 Cubic Class Reference

`#include <Cubic.h>`

Inheritance diagram for Cubic::



Public Member Functions

- **Cubic** ()
- **Cubic** (**Vector2 cp0**, **Vector2 cp1**, **Vector2** ep)
- virtual void **GetControlPointPointers** (std::vector< **Vector2** * > &op)
- virtual void **Rasterize** (**Vector2** sp, bool drawWireframe, const **Matrix2x3** &userToPixel)
- virtual **Segment** * **Clone** ()
- void **ApproximateWithRasterizable** (std::list< **Segment** * > &res, **Vector2** sp, const **Matrix2x3** &userToPixel)
- float **GetMaxRasterizationError** (**Vector2** spPixel, **Vector2** cp0Pixel, **Vector2** cp1Pixel, **Vector2** epPixel)

Static Public Member Functions

- void **Init** ()

Public Attributes

- **Vector2 cp0**
- **Vector2 cp1**

5.2.1 Detailed Description

The cubic bezier curve segment type

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Cubic::Cubic () `[inline]`

The cubic bezier curve segment type constructor

5.2.2.2 Cubic::Cubic (Vector2 *cp0*, Vector2 *cp1*, Vector2 *ep*) `[inline]`

The cubic bezier curve segment type constructor

Parameters:

cp0 Control point

cp1 Control point

ep End point

5.2.3 Member Function Documentation

5.2.3.1 void Cubic::ApproximateWithRasterizable (std::list< Segment * > & *res*, Vector2 *sp*, const Matrix2x3 & *userToPixel*) [virtual]

Returns a list of segments which are rasterizable, and which approximate the original segment with an error less than maxDistance.

Parameters:

res A list of rasterizable segments. The method adds its output to the end of this list.

sp Starting point of segment

userToPixel Transformation matrix

Reimplemented from **Segment** (p. 30).

5.2.3.2 virtual Segment* Cubic::Clone () [inline, virtual]

Returns a new clone of this segment

Reimplemented from **Segment** (p. 30).

5.2.3.3 virtual void Cubic::GetControlPointPointers (std::vector< Vector2 * > & *op*) [inline, virtual]

Adds pointers to the control points to the specified vector.

Parameters:

op Pointers to the segment's control points are added to this vector

Reimplemented from **Segment** (p. 30).

5.2.3.4 float Cubic::GetMaxRasterizationError (Vector2 *spPixel*, Vector2 *cp0Pixel*, Vector2 *cp1Pixel*, Vector2 *epPixel*)

Returns the maximum rasterization error for the cubic curve.

Parameters:

spPixel Starting point in surface coordinates (pixel units)

cp0Pixel Control point in surface coordinates

cp1Pixel Control point in surface coordinates

epPixel End point in surface coordinates

5.2.3.5 void Cubic::Init () [static]

Initializes the cubic curve class. Must be called at the start of the application.

5.2.3.6 void Cubic::Rasterize (Vector2 *sp*, bool *drawWireframe*, const Matrix2x3 & *userToPixel*) [virtual]

Rasterizes the segment. Must first call PrepareForRender successfully to calculate t0, t1, t2 and t3.

Parameters:

sp Starting point of segment

drawWireframe Set this to true to draw a wireframe visualization

userToPixel Transformation matrix

Reimplemented from **Segment** (p. 30).

5.2.4 Member Data Documentation

5.2.4.1 Vector2 Cubic::cp0

Control points of segment

5.2.4.2 Vector2 Cubic::cp1

Control points of segment

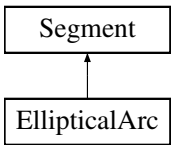
The documentation for this class was generated from the following files:

- Cubic.h
- Cubic.cpp

5.3 EllipticalArc Class Reference

`#include <EllipticalArc.h>`

Inheritance diagram for EllipticalArc::



Public Member Functions

- **EllipticalArc** ()
- **EllipticalArc** (const **Vector2** r, float rot, const **Vector2** ep, int sel)
- virtual void **GetControlPointPointers** (std::vector< **Vector2** * > &op)
- virtual **Segment** * **Clone** ()
- void **ApproximateWithRasterizable** (std::list< **Segment** * > &res, **Vector2** sp, const **Matrix2x3** &userToPixel)
- virtual void **Rasterize** (**Vector2** sp, bool drawWireframe, const **Matrix2x3** &userToPixel)

Public Attributes

- **Vector2 r**
- float **rot**
- int **sel**

5.3.1 Detailed Description

The elliptical arc segment type

5.3.2 Constructor & Destructor Documentation

5.3.2.1 EllipticalArc::EllipticalArc () `[inline]`

The elliptical arc segment type constructor

5.3.2.2 EllipticalArc::EllipticalArc (const Vector2 *r*, float *rot*, const Vector2 *ep*, int *sel*) `[inline]`

The elliptical arc segment type constructor

Parameters:

r Horizontal and vertical radius

rot Rotation angle (in radians)

ep End point

sel Select one of four possible arcs, range 0-3

5.3.3 Member Function Documentation

5.3.3.1 void EllipticalArc::ApproximateWithRasterizable (std::list< Segment * > & *res*, Vector2 *sp*, const Matrix2x3 & *userToPixel*) [inline, virtual]

Returns a list of segments which are rasterizable, and which approximate the original segment with an error less than maxDistance.

Parameters:

res A list of rasterizable segments. The method adds its output to the end of this list.

sp Starting point of segment

userToPixel Transformation matrix

Reimplemented from **Segment** (p. 30).

5.3.3.2 virtual Segment* EllipticalArc::Clone () [inline, virtual]

Returns a new clone of this segment

Reimplemented from **Segment** (p. 30).

5.3.3.3 virtual void EllipticalArc::GetControlPointPointers (std::vector< Vector2 * > & *op*) [inline, virtual]

Adds pointers to the control points to the specified vector.

Parameters:

op Pointers to the segment's control points are added to this vector

Reimplemented from **Segment** (p. 30).

5.3.3.4 virtual void EllipticalArc::Rasterize (Vector2 *sp*, bool *drawWireframe*, const Matrix2x3 & *userToPixel*) [inline, virtual]

Rasterizes the segment.

Parameters:

sp Starting point of segment

drawWireframe Set this to true to draw a wireframe visualization

userToPixel Transformation matrix

Reimplemented from **Segment** (p. 30).

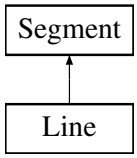
The documentation for this class was generated from the following file:

- EllipticalArc.h

5.4 Line Class Reference

`#include <Line.h>`

Inheritance diagram for Line::



Public Member Functions

- **Line** (const **Vector2** ep)
- virtual void **GetControlPointPointers** (std::vector< **Vector2** * > &op)
- virtual **Segment** * **Clone** ()
- void **ApproximateWithRasterizable** (std::list< **Segment** * > &res, **Vector2** sp, const **Matrix2x3** &userToPixel)
- virtual void **Rasterize** (**Vector2** sp, bool drawWireframe, const **Matrix2x3** &userToPixel)

5.4.1 Detailed Description

The Line segment type

5.4.2 Constructor & Destructor Documentation

5.4.2.1 Line::Line (const Vector2 *ep*) `[inline]`

The Line segment type constructor

Parameters:

ep End point

5.4.3 Member Function Documentation

5.4.3.1 void Line::ApproximateWithRasterizable (std::list< Segment * > & *res*, Vector2 *sp*, const Matrix2x3 & *userToPixel*) `[inline, virtual]`

Returns a list of segments which are rasterizable, and which approximate the original segment with an error less than maxDistance.

Parameters:

res A list of rasterizable segments. The method adds its output to the end of this list.

sp Starting point of segment

userToPixel Transformation matrix

Reimplemented from **Segment** (p. 30).

5.4.3.2 virtual Segment* Line::Clone () [inline, virtual]

Returns a new clone of this segment

Reimplemented from **Segment** (p. 30).

5.4.3.3 virtual void Line::GetControlPointPointers (std::vector< Vector2 * > & *op*) [inline, virtual]

Adds pointers to the control points to the specified vector.

Parameters:

op Pointers to the segment's control points are added to this vector

Reimplemented from **Segment** (p. 30).

5.4.3.4 virtual void Line::Rasterize (Vector2 *sp*, bool *drawWireframe*, const Matrix2x3 & *userToPixel*) [inline, virtual]

Rasterizes the segment.

Parameters:

sp Starting point of segment

drawWireframe Set this to true to draw a wireframe visualization

userToPixel Transformation matrix

Reimplemented from **Segment** (p. 30).

The documentation for this class was generated from the following file:

- Line.h

5.5 Matrix2x3 Class Reference

`#include <vecmath.h>`

Public Member Functions

- **Matrix2x3** (const **Matrix2x3** &op)
- void **makeScale** (const **Vector2** op)
- void **operator=** (const **Matrix2x3** &op)
- void **makeIdentity** ()
- **Vector2 operator** * (const **Vector2** &op) const
- const float & **operator[]** (int i) const
- float & **operator[]** (int i)

5.5.1 Detailed Description

2x3 matrix class

The documentation for this class was generated from the following file:

- vecmath.h

5.6 Path Class Reference

`#include <Path.h>`

Public Member Functions

- **Path** ()
- **Path** (const std::string &str)
- void **GetControlPointPointers** (std::vector< **Vector2** * > &op)
- void **Draw** (bool drawWireframe, const **Matrix2x3** &userToPixel)

Static Public Member Functions

- void **Init** ()

Public Attributes

- **Vector3 fillColor**
- **Vector3 strokeColor**
- bool **fill**
- bool **stroke**
- int **fillRule**
- float **strokeWidth**
- std::string **name**
- std::vector< **Subpath** > **subpaths**

5.6.1 Detailed Description

The Path class. Defines a shape consisting of any number of subpaths. Specifies some rendering parameters such as color.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 Path::Path () `[inline]`

A Path constructor.

5.6.2.2 Path::Path (const std::string & *str*) `[inline]`

A Path constructor.

Parameters:

str The name of the path

5.6.3 Member Function Documentation

5.6.3.1 void Path::Draw (bool *drawWireframe*, const Matrix2x3 & *userToPixel*)

Draw the path

Parameters:

drawWireframe Use a wireframe debug rendering mode

userToPixel Transformation matrix

5.6.3.2 void Path::GetControlPointPointers (std::vector< Vector2 * > & *op*)

Adds pointers to the control points to the specified vector.

Parameters:

op Pointers to all the segments' control points are added to this vector

5.6.3.3 void Path::Init () [static]

Initialize rendering system. This calls the segment types' init functions.

The documentation for this class was generated from the following files:

- Path.h
- Path.cpp
- Path_draw.cpp

5.7 Poly Class Reference

`#include <Poly.h>`

Public Member Functions

- **Vector2 GetCentroid** ()
- void **RenderSimpleFan** (const **Matrix2x3** &userToPixel)
- void **RenderDividingTriangle** (const **Matrix2x3** &userToPixel, bool openPath)

Public Attributes

- std::vector< **Vector2** > **points**

5.7.1 Detailed Description

Class representing a polygon.

5.7.2 Member Function Documentation

5.7.2.1 Vector2 Poly::GetCentroid ()

Calculate the centroid of the polygon.

Returns:
The centroid of the polygon

5.7.2.2 void Poly::RenderDividingTriangle (const Matrix2x3 & *userToPixel*, bool *openPath*)

Rasterize the polygon using the stencil algorithm with my dividing triangle approach for triangulation.

Parameters:
userToPixel Transformation matrix
openPath Optimize for the case where the first and last point of the polygon are assumed to be located far from each other.

5.7.2.3 void Poly::RenderSimpleFan (const Matrix2x3 & *userToPixel*)

Rasterize the polygon using a traditional version of the stencil algorithm where a triangle fan is drawn towards the centroid.

Parameters:
userToPixel Transformation matrix

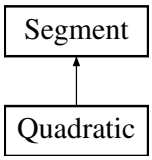
The documentation for this class was generated from the following files:

- Poly.h
- Poly.cpp

5.8 Quadratic Class Reference

`#include <Quadratic.h>`

Inheritance diagram for Quadratic::



Public Member Functions

- **Quadratic** ()
- **Quadratic** (**Vector2** cp, **Vector2** ep)
- virtual void **GetControlPointPointers** (std::vector< **Vector2** * > &op)
- virtual void **ApproximateWithRasterizable** (std::list< **Segment** * > &res, **Vector2** sp, const **Matrix2x3** &userToPixel)
- virtual void **Rasterize** (**Vector2** sp, bool drawWireframe, const **Matrix2x3** &userToPixel)
- virtual **Segment** * **Clone** ()

Static Public Member Functions

- void **Init** ()
- float **GetMaxRasterizationError** (**Vector2** spPixel, **Vector2** cpPixel, **Vector2** epPixel)

Public Attributes

- **Vector2 cp**

5.8.1 Detailed Description

The quadratic bezier curve segment type

5.8.2 Constructor & Destructor Documentation

5.8.2.1 Quadratic::Quadratic () `[inline]`

The quadratic bezier curve segment type constructor

5.8.2.2 Quadratic::Quadratic (Vector2 *cp*, Vector2 *ep*) `[inline]`

The quadratic bezier curve segment type constructor

Parameters:

cp Control point

ep End point

5.8.3 Member Function Documentation

5.8.3.1 void Quadratic::ApproximateWithRasterizable (std::list< Segment * > & *res*, Vector2 *sp*, const Matrix2x3 & *userToPixel*) [virtual]

Returns a list of segments which are rasterizable, and which approximate the original segment with an error less than maxDistance.

Parameters:

res A list of rasterizable segments. The method adds its output to the end of this list.

sp Starting point of segment

userToPixel Transformation matrix

Reimplemented from **Segment** (p. 30).

5.8.3.2 virtual Segment* Quadratic::Clone () [inline, virtual]

Returns a new clone of this segment

Reimplemented from **Segment** (p. 30).

5.8.3.3 virtual void Quadratic::GetControlPointPointers (std::vector< Vector2 * > & *op*) [inline, virtual]

Adds pointers to the control points to the specified vector.

Parameters:

op Pointers to the segment's control points are added to this vector

Reimplemented from **Segment** (p. 30).

5.8.3.4 float Quadratic::GetMaxRasterizationError (Vector2 *spPixel*, Vector2 *cpPixel*, Vector2 *epPixel*) [static]

Returns the maximum rasterization error for the cubic curve.

Parameters:

spPixel Starting point in surface coordinates (pixel units)

cp0Pixel Control point in surface coordinates

cp1Pixel Control point in surface coordinates

epPixel End point in surface coordinates

5.8.3.5 void Quadratic::Init () [static]

Initializes the quadratic curve class. Must be called at the start of the application.

5.8.3.6 void Quadratic::Rasterize (Vector2 *sp*, bool *drawWireframe*, const Matrix2x3 & *userToPixel*) [virtual]

Rasterizes the segment.

Parameters:

sp Starting point of segment

drawWireframe Set this to true to draw a wireframe visualization

userToPixel Transformation matrix

Reimplemented from **Segment** (p. 30).

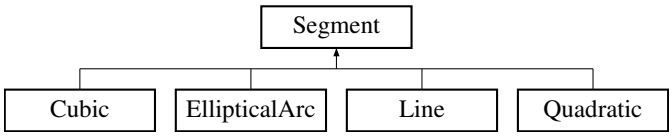
The documentation for this class was generated from the following files:

- Quadratic.h
- Quadratic.cpp

5.9 Segment Class Reference

`#include <Path.h>`

Inheritance diagram for Segment::



Public Member Functions

- **Segment** (ESegType type)
- **Segment** (ESegType type, const **Vector2** ep)
- virtual **Segment** * **Clone** ()
- virtual void **GetControlPointPointers** (std::vector< **Vector2** * > &op)
- virtual void **ApproximateWithRasterizable** (std::list< **Segment** * > &res, **Vector2** sp, const **Matrix2x3** &userToPixel)
- virtual void **Rasterize** (**Vector2** sp, bool drawWireframe, const **Matrix2x3** &userToPixel)

Public Attributes

- ESegType **type**
- **Vector2 ep**

5.9.1 Detailed Description

The base class of all the segment types

5.9.2 Constructor & Destructor Documentation

5.9.2.1 Segment::Segment (ESegType *type*) `[inline]`

The base class constructor

Parameters:

type The type of segment.

5.9.2.2 Segment::Segment (ESegType *type*, const Vector2 *ep*) `[inline]`

The base class constructor

Parameters:

ep End point of segment

type The type of segment.

5.9.3 Member Function Documentation

5.9.3.1 virtual void Segment::ApproximateWithRasterizable (std::list< Segment * > & *res*, Vector2 *sp*, const Matrix2x3 & *userToPixel*) [virtual]

Returns a list of segments which are rasterizable, and which approximate the original segment with an error less than maxDistance.

Parameters:

res A list of rasterizable segments. The method adds its output to the end of this list.

sp Starting point of segment

userToPixel Transformation matrix

Reimplemented in **Cubic** (p. 16), **EllipticalArc** (p. 19), **Line** (p. 20), and **Quadratic** (p. 27).

5.9.3.2 virtual Segment* Segment::Clone () [virtual]

Returns a new clone of the segment

Reimplemented in **Cubic** (p. 16), **EllipticalArc** (p. 19), **Line** (p. 21), and **Quadratic** (p. 27).

5.9.3.3 virtual void Segment::GetControlPointPointers (std::vector< Vector2 * > & *op*) [inline, virtual]

Adds pointers to the control points to the specified vector.

Parameters:

op Pointers to the segment's control points are added to this vector

Reimplemented in **Cubic** (p. 16), **EllipticalArc** (p. 19), **Line** (p. 21), and **Quadratic** (p. 27).

5.9.3.4 virtual void Segment::Rasterize (Vector2 *sp*, bool *drawWireframe*, const Matrix2x3 & *userToPixel*) [virtual]

Rasterizes the segment.

Parameters:

sp Starting point of segment

drawWireframe Set this to true to draw a wireframe visualization

userToPixel Transformation matrix

Reimplemented in **Cubic** (p. 17), **EllipticalArc** (p. 19), **Line** (p. 21), and **Quadratic** (p. 28).

The documentation for this class was generated from the following file:

- Path.h

5.10 Stats Struct Reference

`#include <Stats.h>`

Public Member Functions

- void **Print** ()

Public Attributes

- int **polyCount**
- int **segmentCount**
- int **tileListCmdCount**
- std::set< **Vector2** > **vertices**

5.10.1 Detailed Description

The class Stats contains data that can be collected over a period of time.

5.10.2 Member Function Documentation

5.10.2.1 void Stats::Print () [inline]

Prints collected statistics to the console window

The documentation for this struct was generated from the following file:

- Stats.h

5.11 Subpath Class Reference

`#include <Path.h>`

Public Member Functions

- **Subpath** (const **Subpath** &op)
- void **operator=** (const **Subpath** &op)
- virtual ~**Subpath** ()
- **Subpath** ()
- void **GetControlPointPointers** (std::vector< **Vector2** * > &op)
- **Vector2 GetStartingPoint** ()
- **Vector2 GetCentroid** ()
- void **RasterizeIntoStencil** (bool drawWireframe, const **Matrix2x3** &userToPixel)
- **Subpath ApproximateWithRasterizable** (const **Matrix2x3** &userToPixel)

Public Attributes

- bool **openPath**
- **Vector2 openPathSp**
- std::list< **Segment** * > **segs**

5.11.1 Detailed Description

The subpath class. Defines a shape which is part of a path using an array of segments.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 Subpath::Subpath (const Subpath & *op*)

A subpath copy constructor.

5.11.2.2 Subpath::~Subpath () `[virtual]`

A subpath deconstructor. Note: Deletes all segments pointed to by elements of the segs list.

5.11.2.3 Subpath::Subpath () `[inline]`

A subpath constructor.

5.11.3 Member Function Documentation

5.11.3.1 Subpath Subpath::ApproximateWithRasterizable (const Matrix2x3 & *userToPixel*)

Returns a subpath which contains only segments that are rasterizable, and which approximate the original segment with an error less than maxDistance.

Parameters:
userToPixel Transformation matrix

Returns:
Rasterizable subpath

5.11.3.2 Vector2 Subpath::GetCentroid ()

Calculate the centroid of the interior polygon

Returns:
The centroid of the interior polygon

5.11.3.3 void Subpath::GetControlPointPointers (std::vector< Vector2 * > & *op*)

Adds pointers to the control points to the specified vector.

Parameters:
op Pointers to the subpath's segments' control points are added to this vector

5.11.3.4 void Subpath::RasterizeIntoStencil (bool *drawWireframe*, const Matrix2x3 & *userToPixel*)

Rasterizes the subpath into the stencil buffer using a variant of Kokojima et al's approach

The documentation for this class was generated from the following files:

- Path.h
- Path.cpp
- Path_draw.cpp

5.12 Vector2 Struct Reference

`#include <vecmath.h>`

Public Member Functions

- **Vector2** (float x, float y)
- **Vector2 operator-** (const **Vector2** &op) const
- **Vector2 operator**+ (const **Vector2** &op) const
- **Vector2 operator** ∗ (float op) const
- void **operator**+= (const **Vector2** &op)
- void **operator-**= (const **Vector2** &op)
- void **operator** ∗= (float op)
- bool **operator!**= (const **Vector2** &op)
- bool **operator**== (const **Vector2** &op)
- float **magnitudeSquared** () const
- float **magnitude** () const
- float **dot** (**Vector2** &op) const
- float **cross** (**Vector2** &op) const
- **Vector2 normalize** () const
- bool **operator**< (const **Vector2** &op) const

Public Attributes

- float **x**
- float **y**

5.12.1 Detailed Description

Two-dimensional vector class

The documentation for this struct was generated from the following file:

- vecmath.h

5.13 Vector3 Class Reference

`#include <vecmath.h>`

Public Member Functions

- **Vector3** (float x, float y, float z)
- float **magnitude_pow** () const
- float **magnitude** () const
- **Vector3 normalize** () const
- **Vector3 operator** * (float op) const
- **Vector3 operator-** (const **Vector3** &op) const
- **Vector3 operator**+ (const **Vector3** &op) const
- bool **operator**== (const **Vector3** &op) const
- bool **operator**< (const **Vector3** &op) const
- **Vector3 cross** (const **Vector3** &op) const
- float **dot** (const **Vector3** &op)

Public Attributes

- float **x**
- float **y**
- float **z**

5.13.1 Detailed Description

Three-dimensional vector class

The documentation for this class was generated from the following file:

- vecmath.h

Index