The Norwegian University of Science and Technology

Visualizing and Mapping Rendra's 3D BIM Models to the Real World Using Augmented Reality

Professional programming - Course Report

Authors
Bjarte Klyve Larsen
Morten Omholt-Jensen
Jørgen Hanssen

Programming
Department of Computer Technology and Informatics
The Norwegian University of Science and Technology, Gjøvik, Norway

Contents

1 Introduction

The BIM standard is becoming increasingly popular in the construction industry, and it is therefore essential for the workers on site to be able to use these models. StreamBIM is a mobile/web-based BIM app developed by Rendra AS and is utilized by construction workers on a variety of different projects. StreamBIM enables users to visualize and stream BIM models amongst many other features.

In our thesis, we are ascertaining how implementing Augmented Reality (AR) can benefit StreamBIM and make it simpler for users to work with BIM models creatively.

1.1 Project Description

A finished AR feature would be implemented in the already existing StreamBIM app; however, for this thesis, we are developing a separate app solely for AR. The reason for this is that the current platform uses web technologies, which is suitable for StreamBIM's intended use, but an AR implementation requires a native implementation, which also boosts the performance and creates a better experience for the end user. The app is being developed for both Android and iOS and uses StreamBIM’s existing infrastructure and data to render buildings over the real world in AR. There are a bunch of existing tools to make the implementation more manageable, and we chose to use ARKit and ARCore for this project.

1.2 Project Organization

1.2.1 Group Organization

Due to us being three developers working on two different platforms and codebases, we split the group into two teams; the Android team and the iOS team. The Android team consists of Hanssen and Omholt-Jensen, and the iOS team consists of Larsen. We decided on this because of each developer’s experiences with the platforms, as well as each platform’s complexity; Android is addressed more developers due to being the most complicated platform.

1.2.2 Work Model

The work-model we chose to work with was Scrum with sprints lasting two weeks. At the start of a sprint, we prepare the sprint's work by selecting tasks from the backlog and estimating each task's necessary work. However, a sprint’s tasks are not static, which means we can add tasks as we go if we finish the tasks faster. A good sprint for us will have some tasks left over since this indicates that we have been productive as long as there are not too many unfinished tasks. At the end of each sprint, we conduct a retrospective meeting which allows us to go through our execution of the sprint and highlight the good, the bad, and any potential improvements for the next sprint. We also visit Rendra in Oslo to present the completed sprint's work.

Furthermore, we conduct daily standups early each workday; allowing us to catch up to each other and remove any blockages. We also implemented the spike pattern to give us extra material to write about in the thesis. With spikes, every task that requires any research should have a small research paper written with discussions and comparisons with other similar solutions.

1.2.3 Architecture

We follow best practices for both platforms and try to keep the code and structure as similar as possible without breaking the best practices set. The architecture chosen for each platform was well defined and researched before we started the project. We created a set of rules to follow during development to ensure that the end product was compliant to the spec defined in the project plan.

Android
Apps developed for Android are encouraged to follow a predefined best-practice pattern, which we follow alongside the MVC pattern to ensure that we are in spec and that onboarding new developers will be easy. The android spec requires developers to follow a pattern that splits the entire application up into smaller testable parts, which allows for simple integration- and functionality tests at a later stage.

iOS
The iOS platform also uses the MVC pattern but splits the modules differently to ease testing and development. Also, there are no set guidelines on how to structure an iOS project, which is determined by the developer. There are no clear winners when it comes to how to structure the iOS application; the architecture was defined and created based on prior experience with iOS development. We modeled this to follow as closely to the Android spec as we could while still keeping it “Swifty” in nature.

1.2.4 Development Environment

Version Management

We chose to use GIT as our version control system, which is the industry standard and allowed us to use particular tools alongside the development process. We run pre-commit hooks running our linting tools to provide a consistent code layout alongside the tests making sure that everything passes before developers are allowed to push something to the remote repository. By default, the master repository is set as a protected branch. By doing this, nothing is pushed directly to master but added using pull requests from each feature branch. The base branch is development and is the one we use when working on sprints. Master thus being protected to releases, and only being pushed to at the end of the sprint. Alongside git we use a pattern called GitFlow; this allows a cleaner branch structure which divides the branches into a specific role, such as feature, bugfix or hotfix.

Development Tools

For Android development, we used Android Studio alongside Kotlin. Kotlin was chosen as a language because it is more similar to Swift, thus making code sharing more straightforward between the two platforms. Ktlint was chosen as the linting tool and is enforced with a pre-commit hook.

For iOS Development XCode was chosen alongside Swift. Swift is faster and easier to use than Objective-C. SwiftLint is the lining tool chosen for swift and is enforced with a pre-commit hook.

We have tested the possibility of running a react-native interface on top of our existing native layers, which will enable us to create a consistent user experience on both platforms. Although this is outside of the project scope, it is something we will work on if we have the time to do so.

1.3 Project Plan and Actual Milestones

Milestone	Description	Tasks	Sprint	Significance
Project Plan	An overview of the project and its execution.	Planning Writing	1	Critical
Rendering Engine	Intermediate layers between SceneKit and Sceneform for rendering 3D models on each of their respective platforms.	Download and parse manifest and octrees Build 3D objects from octrees Implement Sceneform and SceneKit for their respective platforms Render the 3D objects in a scene	1 - 2	Critical
Thesis	The document submitted for candidature for an academic degree.	Document pre-development research Document development observations Compose final report	2 - 6	Critical
Code Documentation	General code documentation for the bachelor thesis and for Rendra to use for later development.	Continuously document code	2 - 6	Major
Display 3D in AR	Use models from rendering engines to display 3D objects in AR.	Find walls in the current room Measure distance between corners Match measurement against 3D objects Render 3D objects in AR Scene	3 - 4	Critical
Camera Movement	Rendering scenes based on camera properties and movement.	Anchor the 3D objects to the corners and keep the closest objects in memory	4 - 5	Major
Anchoring and Movement	Generate anchor points for ARK models thus enabling the model's alignment while moving the camera	Find corners Match corners to 3D objects Create anchor points	4 - 5	Major
Layers	Ability to toggle between different BIM structural layers.	Toggle between different BIM model layers	5	Minor

2 Reflections on the Development Process

2.1 Perspective #1: Choice of the Rendering Method

Our original planned method of rendering was to develop custom rendering engines from scratch for both platforms. However, we deviated from our original plan and decided to use existing native rendering libraries, mainly Scenekit for iOS and Sceneform for Android. These libraries integrate well with the AR libraries for their respective platforms and save us time which we could spend on more critical problems such as the AR implementation itself.

2.1.1 Overall Reflection and Experience

We started the development process by developing a rendering engine from scratch in iOS using Metal and Swift during a hackathon hosted by the "Rapid Prototyping" class. The hackathon gave valuable insight into how rendering works, the data formats we would need to account for from StreamBIM, and if a custom rendering engine would be a viable option.

After the hackathon, we were able to render 3D objects using SceneKit and our custom rendering engine. Knowing nothing about rendering beforehand, we made some very obvious mistakes with our rendering engine, yet still, we realized how much more work it would take and how much time we had to spend repeating the process for Android. Along with the time it would take developing a smooth transition to AR from the 3D rendering environment, we decided that two custom rendering engines were far enough out of our AR specific scope to implement in this project.

2.1.2 Existing Rendering Engines

After having experienced the development process of creating a rendering engine from scratch it would be useful to analyze already existing rendering engines and whether or not they would suit our needs. The main problem or skepticism we had when it came to using existing rendering engines is that they are primarily suitable for games. This would not necessarily have been a huge issue, but it would entail building on top of a very large and complex rendering engine, with various features we would not have any use of such as different types of lighting, textures, and physics. Therefore it would seem logical to develop a rendering engine from scratch which only handles our specifics needs which can scale and be extended as feature needs arise. Keeping that in mind, there are a few rendering engines which could have been relevant for us to use.

For Android, there would be one obvious choice which we in some ways already use in our current implementation. This is the rendering engine developed by Google called Filament. Filament is a highly performant rendering engine which has a lot of focus on rendering techniques such as lighting, textures, and materials. The reason we are already using this engine on Android is that Sceneform is a high-level scene graph library which builds on top of Filament. Therefore this would be a good choice for us if we

had decided to use a rendering engine. The disadvantages of using this engine instead of using Sceneform is that it would take longer to implement and it would increase the complexity of the project. Sceneform is quite performant, and since it is, in its essence, a high-level API for Filament it performs similarly to the rendering engine.

While Google has been pushing their own rendering engine for Android rendering, Apple has only been recommending SceneKit for iOS rendering. It is possible to use Filament on iOS also but this is very experimental. This is mostly because Apple has dropped all support for OpenGL ES on iOS in favor of their own library called Metal. Excluding Filament, there are a few free options for cross-platform and iOS development such as Antiryad Gx, BatteryTech, Corona SDK and EdgeLib. The main disadvantages with these engines are that they are primarily optimized for game development and as we discussed earlier, this would entail including a large library with a lot of unnecessary features into our project.

If we, at a later date, decide to continue this project and would like to stop using Sceneform and SceneKit, we would look further into other available engines. The discussion then would regard how long it would take implementing a custom rendering engine which both suits our specific needs and can compete, performance wise, with the other available rendering engines. We predict that it would be more advantageous for us, developing a custom engine because it would be more light-weight and customized for our needs than the other.

2.1.3 Experience of Working With Sceneform and Scenekit

Working with Sceneform and Scenekit has been a lot easier and less time-consuming than developing rendering engines from scratch. Scenekit has been working great, is very mature for its intended use, and has been working excellent for our project. Sceneform, on the other hand, seems to be intended for much simpler use cases than ours; Sceneform was initially intended to be a rendering tool for ARCore and was not able to render a 3D environment on its own until late September 2018.

While Sceneform was made for loading static object files, what we needed to achieve was dynamic rendering based on vertices with indices and with material properties. This functionality was poorly documented, and it seemed like no one else in that community had done anything similar to us. We had to figure everything out ourselves using the little documentation that we had available, but we managed to get there in the end. In addition to poor documentation, some features for 3D graphics were missing or not yet implemented in the Sceneform library. Early on, we had several issues regarding free camera movement and other simple 3D graphics functionality, but we managed to get all the functionality we wanted in the end.

The nature of Sceneform and JVM combined results in high-level code complexity which is not a problem for an end user, but our team’s initial requirement was to have some level of code similarity between iOS and Android. The immaturity of Sceneform combined with our specific use case leads to some significant differences in the rendering method on iOS and Android.

Even though Sceneform seems a bit immature for our requirements, we have been able to solve all of these issues in a reasonable manner, but the one problem we have had on Android with Sceneform is the

Android heap size limit. The Android heap as a hard limit which is different across devices, and since we do not have the time to implement streaming in our AR implementation, we have a significant memory problem since we, on both platforms, load entire BIM models in memory. On iOS, apps can allocate most of the memory, but this is sadly not possible on the Android platform.

There are mainly two practical solutions to this problem when developing apps on Android that require a large amount of memory. The first one is to split the rendering into different processes. A process on the Dalvik (Android's VM which executes applications) has a small heap limit, but if we could make several of them to handle the rendering together, we would solve the memory issue. The problem with this solution is how to pass renderable nodes through IPC (Interprocess communication) pipes. For this to be possible, we would have to serialize the Sceneform nodes which would be a huge task because of the complexity of these nodes.

The other popular solution for a fixed heap size is to use Android NDK (Native Development Kit), which gives applications access to unlimited memory, which was our initial plans for rendering. The problem with this solution is that Sceneform implementations do not support the use of NDK. Using NDK to store model data would solve the memory issue, but this would be hard to do alongside Sceneform.

In this stage of the thesis, we will not prioritize fixing the memory issue on Android as we can load in decently sized models, such as the Smaragd building on the NTNU campus. The problem only arises when trying to load in every layer of a complex building. To showcase our AR implementation we do not need to resolve this issue right now, but if there is ample time at the very end of the thesis we will try to implement a viable solution if possible.

2.2 Perspective #2: Minimap and AR Mapping

Our decision to use scene graph libraries for rendering advanced our progress, and within the first week, most of the rendering was complete. However, we used a significant amount of time-solving issues with the minimap and the mapping of models to the real world in AR. Due to these time-consuming issues, our head-start gradually decreased; which is unfortunate, as we have been ahead of the planned schedule throughout most of the project. Nevertheless, this has not been a significant issue yet, as we decided on an agile work methodology that allows us to deviate from the original plan and still keep the end product in mind.

2.2.1 Overall Reflection and Experience

The process of making the minimap and AR mapping work has been tedious and frustrating as they caused unanticipated and time-consuming problems. However, both tasks have advanced significantly and are nearly finished, but they have required much redundant work that we wish we could apply to other tasks instead.

Both tasks seemed pretty straightforward, and we thought that we solved them within a reasonable timeframe. However, our implementations did not work, and days of testing and failing reminded us that debugging such a complex AR application would be extremely tedious.

2.2.2 Implementing a Minimap

Tilemap

A tilemap is a collection of tiles mapped in a specific order. StreamBIM uses their API to get tiles for floors in buildings and then uses the OpenLayers map library to stitch the tiles together and handle navigating using the minimap. In our project, we chose not to use a map library, mainly because OpenLayers does not exist for mobile development and we did not want to include a more extensive map library for handling tiles. Therefore we implemented a tile service on Android and a tile module on iOS which has three main tasks to do. Firstly, they fetch the correct tiles for the current floor in the current building in the project. Then they stitch the tiles together to form a minimap with the tiles. Lastly, they create render nodes with the 2D tilemap which is later used to render the minimap to a scene graph. This part of the minimap implementation was developed in a short time without any significant problems.

Minimap Controlling the 3D Camera

The next step in the process was to enable touch events on the minimap to control the 3D rendering camera, and enabling the 3D camera to move the minimap. The camera implementation on iOS was straight forward, we use events to pass state between the main camera and the minimap camera. These events are handled within the camera controller class itself thus keeping the two cameras separated. To accomplish the same effect on Android, we first had to rewrite the structure of the camera handling differently. Where we before had one rendering manager class which handled the cameras, we now had a camera class which handles the main camera methods along with two subclasses: a *main* (3D) camera class and a *minimap* (2D) camera class.

We implemented the minimap in such a way that the minimap is a SceneView with its dedicated camera. This method allows us to move the minimap camera in relation to the 3D camera instead of moving the minimap in opposite relation to the main camera. This rewrite on Android took a bit longer than expected because of some mathematical and logical errors in the camera translation and rotation handling mainly caused by complicated mathematical concepts such as quaternions.

The next step was to implement methods for handling the translations between the 2D minimap and the 3D main view. Firstly, we tried to implement an algorithm to calculate how much the cameras had to move in relation to the other. The problem was that different floors and different buildings might have different sized minimap tiles, which made the current solution useless as it only worked for two buildings. After a chat with the product owner, they referred us to their current web solution and explained some more endpoints which we could use.

The next solution we tried to implement was using a transformation matrix supplied by StreamBIM's API for each floor. To do this, we had to implement a few mathematical algorithms to use the new matrices. After fixing some logical and mathematical errors, the solution worked across every project as long as the

transformation matrices were correct. The last step was then handling the different resolutions the different buildings' tilemaps had.

The implementation of the minimap took a lot longer than initially planned, which was caused by a few factors: firstly, the first notable rewrite to improve how we handled the cameras took a long time. Next, the solution which did not work across all buildings, and lastly, fixing some mathematical and logical bugs which took a long time because of the difficulty debugging 3D translations.

2.2.3 Implementing World-Aware AR Mapping

AR Mapping

AR mapping is the process of finding a transformation to apply to a rendered model to make it seem in-place with the real world in AR. In our case, we are mapping a room of a rendered BIM model to the same room in the real world; a user can then move throughout a building under construction and see the rendered building overlayed correctly over the unfinished building, which gives the impression of being inside the rendered building. For such a transformation to work, it would need to contain information about how to rotate and translate the rendered model to appear correctly overlayed with the real world.

The only mapping we have implemented up until now has been a direct camera-camera mapping; this means that the user is required to position their phone's camera to the same position as the camera in the "normal rendering" before going into AR mode. This method works but is very inaccurate, and a mapping using information about the real world to align the BIM model would be considerably better. Fortunately for us, ARCore and ARKit both retrieve information about all detected real-world surfaces, and BIM models usually include information about a room's geometry. StreamBIM already parses this geometry on their backend, and it is easily obtainable for us to parse to the same data as the detected surfaces by ARCore and ARKit. From here, we tried several methods of obtaining a transformation using this information.

The Brute-force Approach

This method first aligns the model to the same grid as the real-world, then rotates the model until it finds the most likely rotation based on vectors between the room's corners. It then finds the same corner in the real world and the model and translates the model to match the same corner in the real world. This method is excellent as the user only needs to be in the same room in “normal rendering” before entering AR without needing to worry about camera positions. However, the user also needs to scan multiple surfaces before the transformation could be correct. Also, a room would have to be *unique* enough to only be correct in one orientation; most rectangular rooms have the same shape when rotated 180 degrees.

The Camera-Aware Approach

This method is much like the brute-force method but uses information about the Scene camera and the phone’s camera to determine the model’s rotation. This method rotates the model correctly, but the user is still required to scan multiple surfaces before the transformation could be correct.

The Progressive Approach

This method finds a start-transformation and continuously maps the model as information about the world is gathered. Our start-transformation is based on the first direct camera-camera transformation, and the algorithm then loops over all AR planes and compares them with their nearest BIM model plane. It then matches the rotation and translates the BIM model so that the planes match. This method required us to implement the Observer pattern so that new mappings could be performed when new AR surfaces are detected. Nonetheless, it allows the user to observe the building in AR without having to scan multiple surfaces beforehand.

Addressing the Methods and the Implementation Problem

We considered all the methods and decided that progressive mapping allows for the best user experience as well as being the simplest to implement. Moreover, we discarded the brute-force method due to its high processing cost for complex rooms and its inaccurate rotation.

The main problem with the implementation was that we were unsuccessful in getting it to work, or rather the translation of the model; we implemented all the solutions, but the problem persisted across all. The algorithm consistently found the correct rotation for the building as well as the same corner in both the room's geometry and the real world, but translating the model from point A to point B turned out to be a struggle.

The Debugging Process and Solution

A fundamental difficulty of debugging AR is that it has to be done by using the app, which is very tedious for our specific AR edge-case. The debugging cycle consisted of building a version of the mapping, trying it, then interpreting logs and attempting to fix the problems. The cycle was very time-consuming as much time was spent using the app and interpreting the vague logs containing geometry information.

After a prolonged period of debugging, we are approaching a working mapping using the progressive method. The current solution maps the model correctly to the floor, however, we found that the mapping of walls does not work due to StreamBIM's initial rotation of the building; StreamBIM rotates the model by 180 degrees on the z-axis due to an old implementation mistake. Nevertheless, StreamBIM does not flip the space geometry the same way as the model, which is why the mapping of the floor works and not the walls. A working implementation should be implemented within a short time-period now that we know the root of the problem.

3 Discussions and Conclusions

General Progress vs. Expectations

We are pleased with the project's progression throughout the semester and are comfortable with its current state. The head-start we received when deciding to use existing rendering frameworks has been crucial for the project's current state, as the unexpected problems introduced by the minimap and AR mapping was very time-consuming. As of now, we are expecting a working minimum viable product (MVP) to be finished before the deadline with a substantial margin, giving us time to focus solely on the thesis report.

Problems and solutions

Throughout development, we have had a few problems that have slowed down our overall progress. The two main problems which have caused the most significant setback is the problems regarding the minimap and the AR mapping. Regarding the minimap, we have found a suitable solution for the issues that arose. The main downside with this issue is that it took a long time solving an issue for a feature which is not crucial for the end product. However, we did get an early start so that we did have a buffer for such problems to occur.

AR mapping has proved a difficult problem to solve, and as it is an essential feature for our end product, it has been necessary to find a solution to this issue. Fortunately, we knew that this feature would be a time-consuming problem and we planned consequently; even though it took longer than expected. We are now confident that we have a robust solution for the issue and will have fixed the last remaining bugs before user testing. In retrospect, we are glad that we chose to use existing rendering engines so that we had the extra time to solve the AR mapping. It was also unwise of us to not consider the initial rotation of the building, as well as trying to implement all the different solutions in the hope that one of the inferior methods would work; we should have focused our work on getting the best solution to work.

As described earlier we have a memory issue on Android which is prominent for larger BIM models or phones with limited memory. There are multiple solutions for this, but we have deprioritized the issue temporarily as the current product is working fine with the issue present and our thesis highlights the AR aspect of the product. However, we will be trying to implement a solution for this issue if there is remaining time at the end of the assignment. As for now, we are satisfied with the current state of our application.

Conclusion

Working on this project for a while has given us many learning experiences; it has not all been good, and we have encountered several pitfalls along the way. If we had to do this over, we would most likely end up structuring our applications differently. The Android architecture model is not suited for applications like this and is more focused around common use-cases. We should have implemented a better testing regiment to ensure that the two experiences are equal and spend time on testing features with code and creating end-to-end tests to make sure the programs work as expected cross-platform. The software development process and team communication have been on point. Our main issues were related to Android and the way that Sceneform handles data, looking at this from the outside, rewriting the rendering aspect of the Android application in NDK and C++ would have saved us several problems. SceneKit seems more mature in its feature set than the Android equivalent. The overall experience with ARCore and ARKit has been surprisingly good. We found some publications and tutorials helping us along at the beginning. They both provide a decent interface to work with and iterating over the features was quick.

We got a lot of support from the Rendra staff throughout the development process; they were always available to chat about the features and discuss the implementation, and how to use their backend API. At our regular meetings, we could demo the application and get feedback on the current progress. This in return gave us a technical project owner for the scrum process.

We were pleased with the toolset we chose. Working with git hooks is tedious because they are not committed to the remote repository, but once it was set up, they worked as a safety net and ensured that our code was linted and up to spec. GitFlow gave us the benefit of structure within GIT. Branches were no longer given random names, but instead, they had some meaning behind them. Scrum was important to us, as it gave us the ability to hone in on specific areas within the sprint duration. Looking back we should have gone for one-week sprints instead of two, allowing us to turn the “ship” at a quicker pace.

The last half year and the process we have been through, developing for and researching the construction industry, has left us with a lot of positive experiences that we can bring with us in our careers. We are left with a prototype of an application that could one day be in the hands of construction workers and ease their day-to-day activities.

If we were to do such a project over again; We would spend more time in the planning phase and testing the options we have on the Android platform and improve code sharing the platforms. IOS supports the use of C, and with Android NDK using a low level language that works cross-platform could potentially ease the process. With our main pain points came from memory management and this is something we should have thought more through at the start of the project, and a solution would be NDK. Over all the management of the project has gone well and we chose the correct work methodology and tools. Cross platform AR solutions are not as mature as we need them to be, but i think we worked around the issues we had at the time. As technology advances Flutter or React-Native can be viable in the future, and is definitely worth looking at.