Interim Report

1. Development Stage

Currently, Team 7 has fully implemented functional minimum and nearly completed low target. Certain elements of shadowing and shading yet remain to be fully adjusted, however the fundamental part and most of the advanced features we intended to implement in the shading and shadowing subtask are finished. The screenshots to corroborate this statement are provided below. Furthermore, we implemented some of the features of the desirable target. Finally, in order to fulfill our needs and facilitate the subsequent development process we implemented a map editor.

Below is a layered task breakdown up to desirable target that provides information on the development stage of each feature.

Layered task breakdown	Development stage	Projected hours	Actual hours
Functional minimum			
Game engine components:			
Render queue	completed	10	20
Basic shading	completed	15	5
Collision system	completed	20	40
Camera	completed	10	15
Basic UI	completed	5	10
Game design:			
Basic 3D models	completed	5	5
Basic static level	completed	10	15
Basic GUI	completed	7.5	7.5
Game mechanics:			
Input and controls	completed	5	30
Basic food collection	completed	5	15
Low target			
Game engine components:			
Lighting and shadows	nearly completed	40	50
LAN	completed	50	90
Game design:			
Improved 3D assets	completed	15	40
Improved map design	completed	15	35
Desirable			
Game engine components:			
Sound effects	in development	5	1
Game mechanics:			
Advanced food collection	N/A	7.5	0
Fighting system	in development	25	10

Game design:			
Improved GUI	in development	7.5	15
Extras			
Map editor	completed	N/A	15

2. Challenges and Revisions

During the development process we have encountered a handful of problems which proved to be harder to solve than expected. Furthermore, we had to make revisions and changes to certain aspects of the architecture at various points of the development process.

2.1 Challenges

2. 1. 1. Content Processor

We plan to use various shading techniques for the models and individual mesh parts based on the assigned material. However, Monogame's built-in model processor just attaches the BasicEffect to each model. Given large number of models which we intend to use with different shading techniques, such as normal maps, manually setting up effects (shading techniques) for every model, or a particular mesh of the model, would have been a highly time consuming and tedious task. Thus, we decided to write our own custom model processor.

Given the scarcity of information on how to hook up a custom effect for the model during the build assets stage in the Monogame pipeline, or simply contradicting or outdated information provided in the tutorials made for the older versions, the task of writing a custom model processor turned out to be quite time consuming.

One of the biggest issues that arose at that point was a missing custom effect writer. In order to understand the inner working of the Monogame pipeline we needed to extensively dive into the Monogame GitHub page. Finally, we managed to solve the problem and had to write the following classes to attach the custom effect to a model:

- CustomModelProccessor.cs – implementation of our custom model processor. The processor now could assign multiple textures to our own custom-effect, there is even a way to load additional textures which the OpenAssetImporter was not able to find.
- MaterialProcessorCustomEffect – custom material processor that is invoked from the custom model processor. This class adds additional textures originally discarded in Monogame, such as normal maps and specular maps. A material of type EffectMaterialContent is used in order to store both the texture and the custom effect specified by a developer.
- NormalMappingTextureProcessor.cs – invoked from custom material processor and is intended to use for the normal maps shader.
- CustomModelWriter.cs – custom content type writer. The Monogame pipeline does not contain a content writer for the type of material we are using to store our effec. Therefore, it was necessary to write our own which writes the value of the compiled effect and a dictionary of textures. Luckily, it turned out that built-in Monogame reader EffectMaterialReader is suitable

for our needs to read out this kind of format, which is used in conjunction with our custom writer.

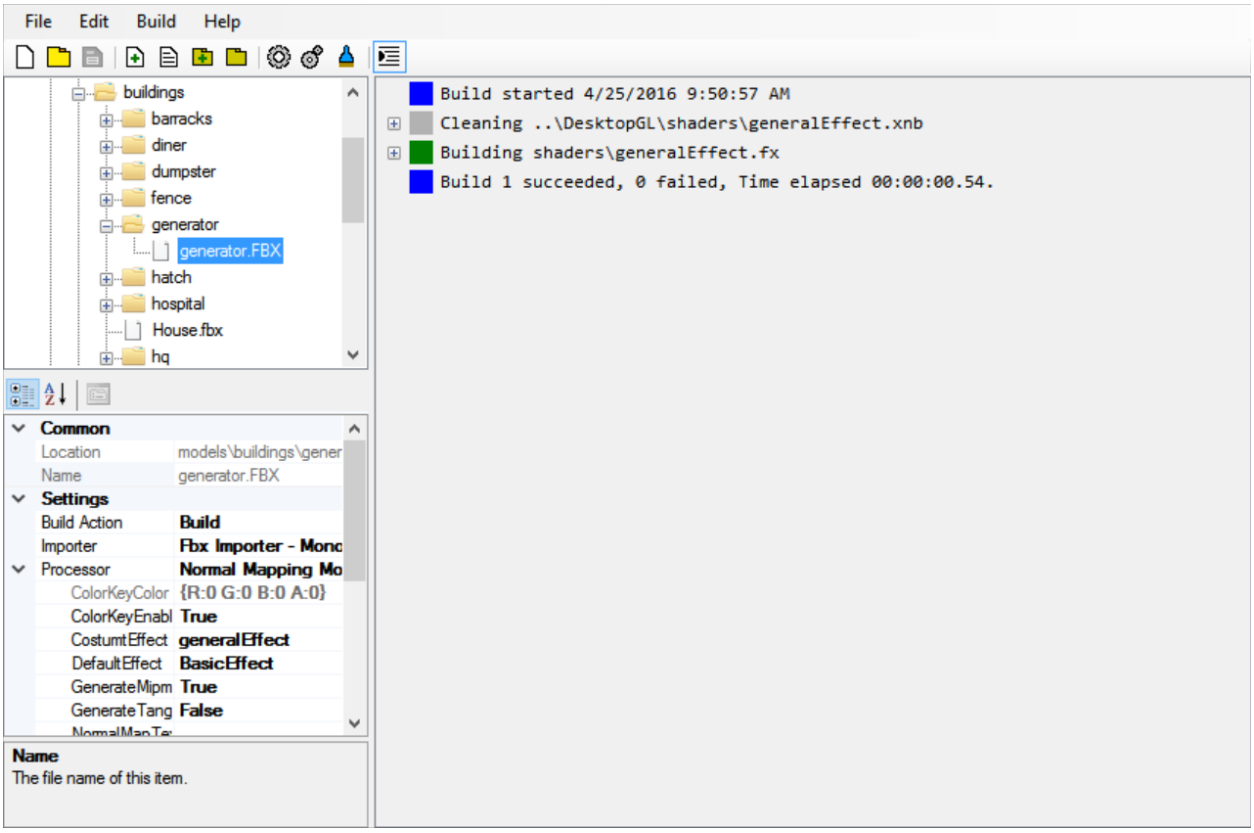

Fig. 2.1. Custom model processor (normal mapping model processor)

Figure 2.1 demonstrates the setup of the custom model processor and an associated custom effect (generalEffect, which has to be put into "Content/shaders" folder).

2. 1. 2. Map Editor

Originally, we designed a static level in 3ds max (fig. 2.2). This meant, that if we use a certain model (e.g. a house) multiple times, its geometric values would also be stored and loaded multiple times. Additionally, exporting one big model for the whole map is not flexible enough and frustum culling would have to be used on every single mesh in the model which is not efficient if a building on the map consists of multiple meshes.

Another downside of this approach was, that only one of us is using 3DS Max and could change the scene. To quickly adjust the map and see how it looks like in the game right away, we needed an editor. An editor allows us to quickly test and balance our maps during development and allows players to create their own maps.

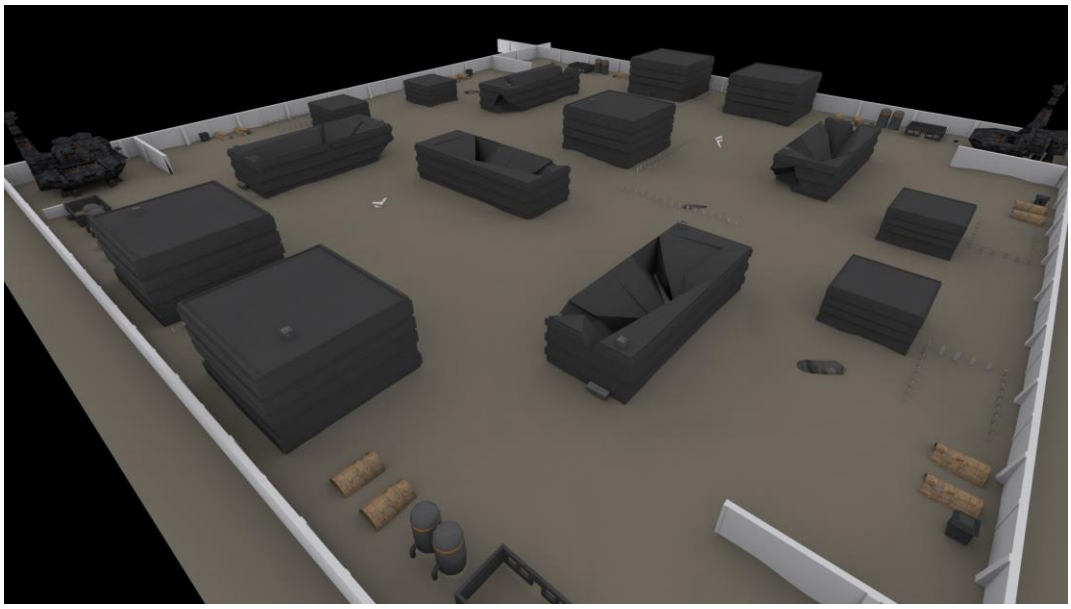

Fig. 2.2. Static level (3ds Max render)

The Map editor dynamically creates categories of objects that can be placed on the map based on our folder structure. It will then assign all models contained in a folder to a category and the user can then easily switch between models in a category using the mouse wheel. This way we don't have to adjust the code of our game if we add models and categories. Figure 2.3 shows the map editor with a diner selected. It can be moved, rotated and placed arbitrarily within the boundaries of the map.

Fig. 2.3. Map editor

2. 1. 3. Advanced Collision System

We realized that the axis aligned bounding boxes provided only a crude approximation of collisions and induce frustration to potential players. In order to mitigate this, issue we realized a two-steps collision detection system:

- Check for the AABB intersection
- Check individual mesh triangles intersection of the dedicated collider meshes of the models

Mesh triangle intersection provide much more realistic approach to collisions and, combined with our frustum culling, it does not weigh heavily on the performance.

2. 1. 4. Custom Shader

In order to make use of various techniques and preserve textures which can be lost when switching from one effect to another, we decided to employ one shader which contains different techniques for different purposes, such as shadow maps, light maps, general shading and map editor shading. The switching between techniques is currently done in the RenderManager class.

2. 1. 5. Lighting

We indent to use multiple lights in the game, such as spot lights for the drones, a point light for the player's immediate area and other light sources spread across the map (street lights, fires, …). Given the fact that HLSL does not support dynamic arrays and, in fact, provides only a limited number of elements in an array, in order to use multiple lights in the scene it is necessary to render each light individually and then blend them all together additively in a single render target. Finally, the render target is converted into a texture and then applied over the whole screen. Fig. 2.4 illustrates that concept. In the left corner of the image the final light map texture is located. The larger portion of the screen shows the final render with the light map texture applied on top of it.

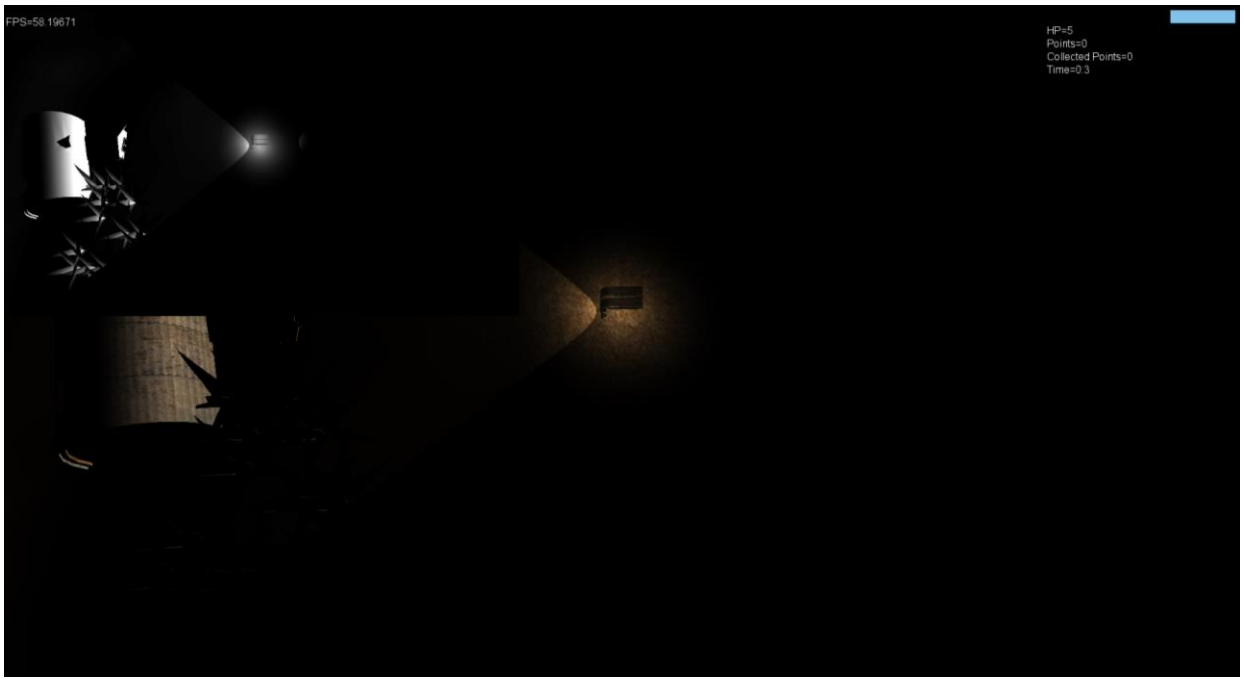

Fig. 2.4. Use of light maps

2. 1. 6. Shadow Maps

Implementing shadow maps proved to be quite challenging. The core idea is easy to implement, but it produces unsatisfactory results which contain so called perspective aliasing (fig. 2.5). The left part of the

image 2.5. represents the depth texture rendered from the perspective of a light source. The right part of the image is the final render with applied shadow maps.

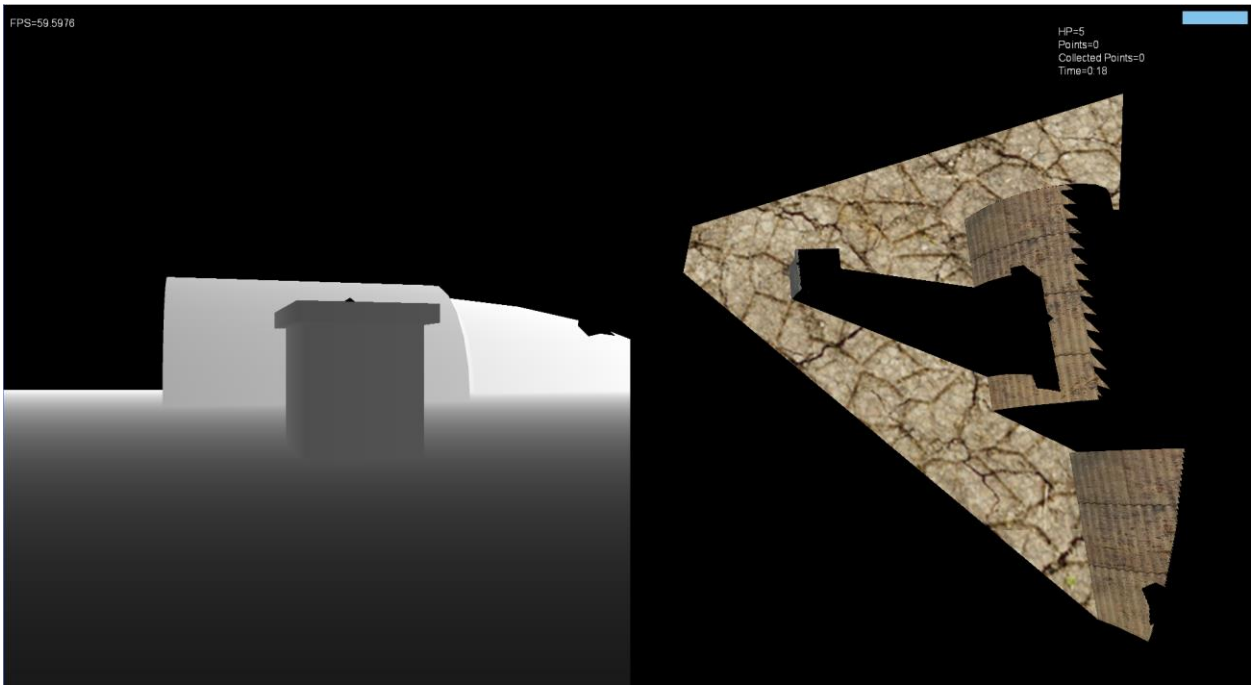

Fig. 2.5. Perspective aliasing

Most of the solutions found on the internet, such as cascade shadow mapping, do not directly address our issue. We were considering making use of cascade shadow mapping at first, but then we realized it won't be helpful since the camera view (eye) has about the same Z distance to all the objects of the environment, thus, we cannot split the frustum into multiple subfrusta.

As of now, we tried to blur the shadows, and it yielded much better visuals, but at the cost of drastic decrease in performance (down to 20 fps on internal GPU, **fig. 2.6**).

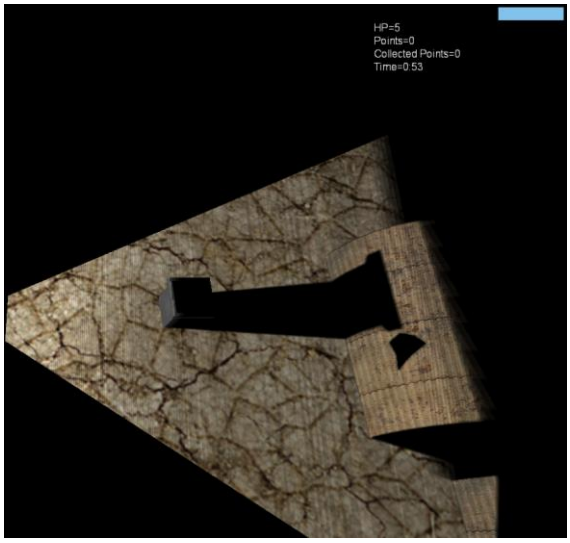

Fig. 2.6. Blurred shadows

2. 1. 7. GUI

We implemented a number of scenes which contain menus. That required us to introduce GUI element classes. Additionally, during this stage of development we also managed to order the draw calls so the GUI elements do not end up hidden behind certain 3D objects, such as ground that covers the whole screen. The GUI elements we added include buttons, text input fields, text and simple images. They can all be individually styled using textures, fonts and colors.

Fig. 2.7. Main menu

2.2 Revisions

Expectedly, the introduction of numerous new features requires revision to the architecture. That is, most of the revision can be boiled down to decoupling certain classes, introducing general super classes or simply adjusting old functions to accommodate changes we made, such as invoking draw function with certain techniques in mind.

3. Conclusion

For us reaching the interim stage mostly incorporated changes that are concealed under the hood, such as a custom model processor and structures that facilitate usage of shading techniques. Compared to the first playable demo the most visible change is the introduction of shadow maps and lighting. Furthermore, we introduced a substantially better collision system and started experimenting with sounds. Moreover, the player is also capable of unloading food supplies at the base. The fighting system was extended, it is now possible to use a shield and enable a temporary speed boost for the drone. Finally, once we adjust the shading techniques (including the introduction of normal maps) we can move on to refining the actual game mechanics and playtesting which will be greatly simplified by our newly created map editor.