Interims Report

1. Functional Minimum

General

Project planning

During the game proposal phase, we were using Microsoft Project (out of curiosity) to create the development schedule. At this point Microsoft Project provided extensive features and functionality to precisely plan the project beforehand. However, it turned out, that firstly each team member needed the program to submit his or her progress. Secondly the extensive feature set was just an overload for the purpose of this project.

Due to this reasons, we created a separate page on the course Wiki and copied the five layers in form of simple tables. This Wiki page gives each team member the possibility to easily keep track of the project status and also to easily submit his or her contribution. It also serves as a persistent place for bug reports and general notes.

For the milestone deliveries the progress and working hours from the Wiki pages will be copied back to the Microsoft Project sheet, since the program can visualize the progress in a quite informative manner.

Project Environment

Already at the beginning of the course, we decided, that we are going to utilize the Unreal Engine 4. The latest version at the start of development was 4.11. The engine is written in C++, but also provides a visual scripting system, called “Blueprints”. However, we want to use the latter one only where it is absolutely necessary and remain mostly with C++, which allows us to further improve our programming skills.

The main target platform is Windows, whereas it is also possible with Unreal Engine to deploy for Linux systems.

Further, we use Visual Studio 15, which is required since this Unreal Engine version, and control our progress with a Git repository on GitLab. Our Git follows no specific workflow, such as Feature branches or Gitflow, since merging the binary asset files of Unreal is a real pain and also quite error-prone.

Gameplay

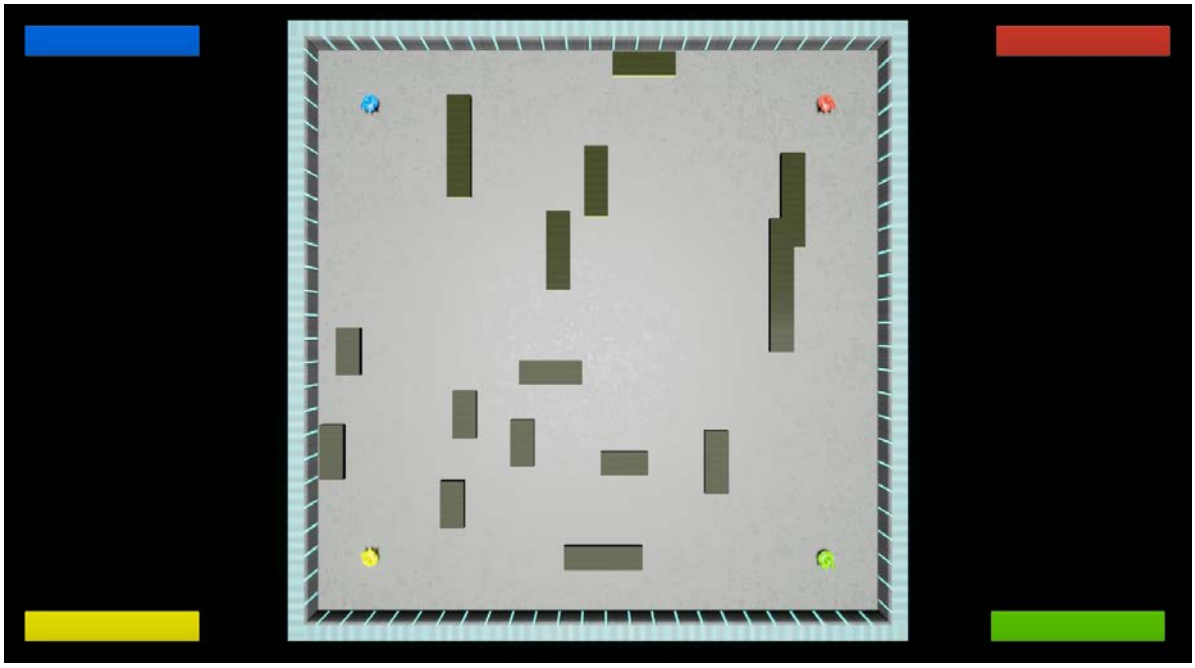
Controls

Unreal Engine comes with a twin-stick control scheme, but it is very basic and insufficient for our purposes, so several adjustments had to be made. First of all, firing has to be controlled by means of a fire button, it is assigned to the right trigger of an Xbox controller. Secondly the player model has to face the direction of the right analog stick, which controls the shooting direction. Ultimately, if the right analog stick isn’t pressed, the player model will face the movement direction again. Regarding this point, we aren’t sure whether the orientation of the player model should rotate back to the movement direction or stay at the shooting direction. This has to be evaluated in the playtesting section and adapted accordingly.

Local Multiplayer

Unreal Engine's native local multiplayer is implemented as a split screen multiplayer with a camera per player. Unfortunately, it is useless for our proposition since we have an arena shooter with one static camera for all players. Firstly, a camera which captures the whole arena has to be added. To replace the player camera, it has to be removed from the player model. Additionally, each player's view target has to be set to the arena camera. To ensure each player spawns when the game starts, custom spawn points with clever spawn logic have to be implemented. So players can only spawn at a point if no other player has spawned there before. Otherwise there will be conflicts and some players won't be added.

In order to distinguish each player form another, each player has a unique ID, which determines the color of the player model.



Obstacles and Arena

The obstacles are composed of several cubes, which are generated with a procedural mesh generator. Therefore, the vertex buffer and index buffer are explicitly defined with a global fixed voxel size. To assure correct UV-mapping, 24 vertices are stored instead of only 8. With those generated cubes – also referred to as voxels in this report -, custom obstacles can be built. At the moment, the obstacle generator provides a function for building rectangles of variable size, but extensions for other shapes can be made.

To be consistent with the obstacles, the arena itself is generated with a procedural mesh generator. The size of the arena is variable and the arena boundaries can also be specified in terms of thickness and depth. The camera depends on the arena size to shows the whole arena.

Projectiles

Projectiles are currently set up by a capsule mesh and its own collision behaviour for the world objects. Each projectile will be shot by the player depending on their weapon heat and energy level. As of now, the projectiles only have one color for testing the changeability of

the material properties. In later builds the color of each projectile will show its energy level from blue (high) to red (low).
If a projectile hits a player, all of its energy will be converted to the damage inflicted on the hit player. Other hit objects will either reflect or refract depending on their refractive index, used for the reflectance coefficient calculated by Schlick's approximation. This gives the percentage of reflectance in contrast to the refraction of certain materials and therefore the energy level of newly created reflected projectiles and the refracted one. The refracted projectile will change its direction by Snell's law, which also takes the refractive index of the two materials on the planar surface into account. The border walls will only reflect and reduce the projectile's energy, so no projectile will leave the arena.
For the interims build, the projectile just deals damage to the players and currently not to the voxels/obstacles, as refractive indices where not yet integrated, but the functionality is already given.

2. Low Target

Gameplay

- The health bars are colored accordingly to the players and represent the health of the respective player.
- If a projectile hits a player, it applies damage to the player according to its current energy level.
- If the health of a player reaches zero, he dies. But since the view targets are bound to the players, simply destroying the player object on death will mess with the camera view. So if a player dies his controls will be disabled and model removed.

Projectiles

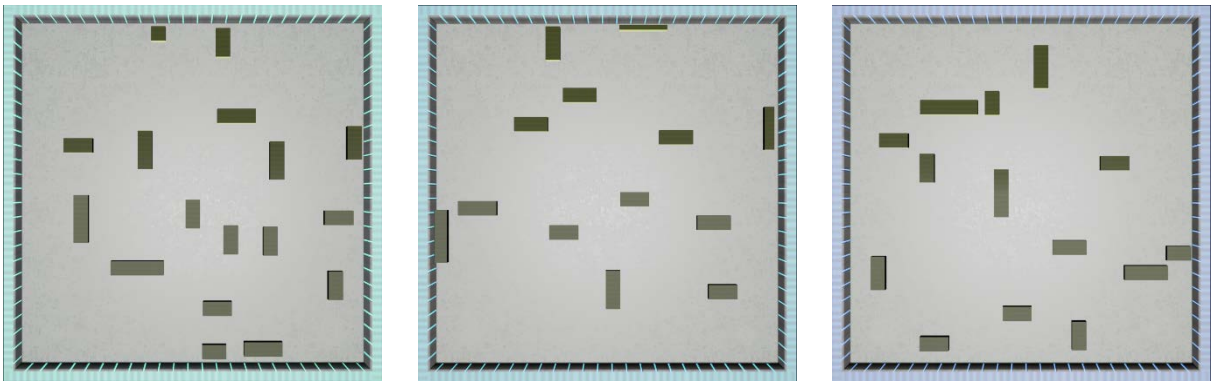
The current projectiles weren't using the correct collision normal for the obstacles, which results in rather strange reflecting directions. For the interim presentation we fell back on a previous build, which used a debugging reflection direction for testing the collision, which simply multiplied the direction by -1, resulting in going back where the projectile came from. Reflections on the border walls where offering the correct normals and intended reflection behaviours. The previous build had a problem with continuous collision detection, which might lead to projectiles ignoring obstacles because of their high speed, and also projectiles being stuck inside an obstacle, as it has registered another collision frame by frame. These problems are already fixed by using the physics system of unreal, which tracks the path of projectiles and therefore checking for collisions in between two frames.
Also the projectiles should have a border color to show, from which player the projectile was being shot. This is not yet really visible because of the speed of projectiles, so this might need some changes or a complete removal to support better understanding and clarity of projectiles in flight.

3. Desirable Target

Gameplay
The menu at the interim state is purely a basic menu with the functionality of playing and quitting the game. However, it provides a starting point for a more advanced menu, which will be completed in the next development phase.

Obstacles
To support destructions, a destruction handler is created. It is able to delete all destroyed voxels of an obstacle and to find separated sup-parts of voxels, of which a new obstacle can be generated. It starts at a random voxel and traverses in a 'first in first out' – queue manner all neighboring voxels, collecting them in a separated array. If all neighbors are visited and there are still unvisited voxels left in the obstacle, then a sub-group was found and a new obstacle needs to be created. This makes it possible to apply physics to the obstacles in a later state of the game development, even after they got destroyed and separated in several pieces. Also, obstacles with a variable height in the z-component, which is not included at the interim report, can be added, since the search algorithm works independently of the axes because it calculates the distances between the center of voxels.
However, in the interims state destructions are theoretically possible, but not included within the game play, since the projectiles do not deal any damage to the obstacles yet.

Random Arena Generator
Random generated arena layouts require the player to adapt to the different obstacle placements in order to fully utilize the light properties of the projectiles. This therefore leads to a higher game depth.
The Random Arena Generator divides the available arena size into N logical axis-aligned cells of random, but limited size. Within each cell one obstacle with random length is placed. Currently, the arena generator limits the obstacles to be either a horizontal or a vertical arrangement of voxels with a defined width.



Projectiles
The advanced reflection for the desirable target includes the usage of refraction and applying damage to crossed voxels inside an obstacle, changing direction of the path by Snell's Law and changing energy level by Schlick's approximation for receiving the reflectance percentage and the remaining refraction energy. What we were missing for the interim build was the change in direction, as obstacles didn't have the refractive index yet, and the change of color of each projectile depending on their energy level.

Projectiles also might need a trail of light following their path to create clarity for the players as they understand the direction of incoming projectiles. As Unreal Engine's particle system is quite complicated, we are currently using a simple, hardly noticeable particle system, which we will change in later builds.

4. Design revisions

While most game elements currently only provide basic functionality, we did not change any aspect of our initial design decisions.

5. Challenges

Using an existing engine allows you to focus on the key elements of game. However, it also requires careful studying of the game engine's architecture to prevent pitfalls. Additionally, you are in some cases limited by the engine's capabilities and design decisions.
The visual scripting system of Unreal Engine provides a fast way to add functionality without any programming. But since we are familiar with programming C++, we do not want to use Blueprints. However, there are some things, which can only be done with Blueprints.