Computer Games Laboratory, Winter Term 2018

Dark Rip

Project Notebook

Moritz Kohr

Phillip Hohenester

Erik Franz

Use With Caution

Contents

1 Formal Game Proposal

1.1 Game Description

In Dark Rip the player controls an asteroid base in orbit around a black hole. They have to control and change their orbit to collect resources to advance their base and use as fuel. Ultimately the player has to collect enough resources to build a device that allows them to escape into the black hole and finish the game.

Game Loop The game loop works as follows:
collect resources, build and protect the asteroid-base, plan base trajectory, execute acceleration maneuvers, collect more resources

1.1.1 Setting

The universe is collapsing towards a last black hole. The last hope of survival is to try and escape annihilation by flying through this black hole and get to the other side were a new universe is said to emerge. The players travels in a spaceship outfitted with a special device which makes this journey possible. But the final necessary calculations will have to be performed on-site. Shortly before they can reach their goal the ship collides with a larger asteroid and is badly damaged. They find that, even though the device was destroyed, the plans survived and there is still enough time to rebuild it and escape the collapse. To that end, they have to establish a base on the asteroid and collect the nearby resources to advance and protect their base while the device is rebuild and the necessary calculations are performed. It's the player's job to coordinate these efforts.

1.1.2 Environment

The game takes place in orbit around a black hole, which marks the center of the playable area. The area around the black hole, beyond its horizon, is the only safe area left, since everything else is already affected by the ongoing collapse. If the base moves too far or too close it will be lost. During the game the collapse continues, closing in on the black hole from all sides, providing the player with an implicit timer while progressively restricting the save space to navigate. The play area is also filled with other asteroids on stable or unstable orbits.

1.1.3 Orbit Mechanics

In the initial version the orbits will be simulated using simple Newtonian physics and a finite steps approach. Later they might be switched over to a Kepler ellipses based version for higher accuracy.

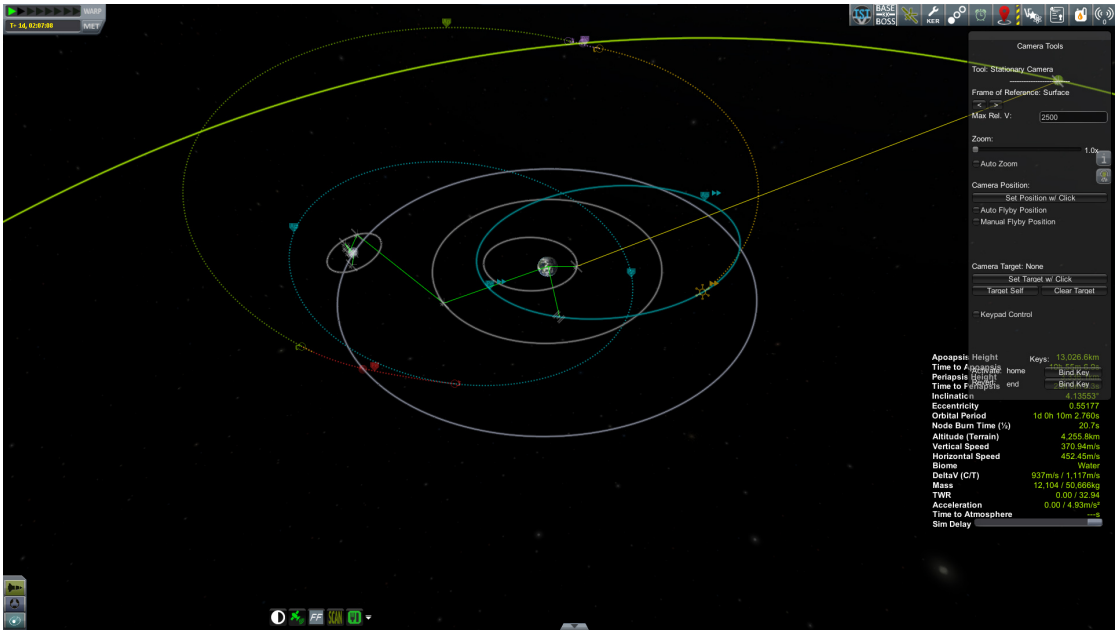

Figure 1: How Kerbal Space Program represents orbits

1.1.4 Asteroid Base

The only entity the player controls is their base. The player can influence its orbit/trajectory by using its propulsion system as well as upgrade or add to its functionality.

Movement/Controls The player can control the base's trajectory by acceleration. Acceleration maneuvers cost mass and/or energy in variable amounts. A planning interface to preview and setup maneuvers similar to Kerbal Space Program is also planned for later in development.

Modules Functional aspects of the player base can be upgraded by constructing additional modules on the asteroid. The initial construction has a cost in mass while existing modules might have a constant energy cost/drain. These are possible types of modules that could be used in the game. The details depend on development progress and balancing.

Module Type	Effect
Propulsion	improves acceleration efficiency and maximum
Mass Collector	increases collection radius and speed threshold
Energy Collector	passively produces energy
Energy Generator	converts mass to energy
Storage	more capacity for mass and energy
Jump Device	needed to finish the game
Research	needed to finish the game
Defense	-
Sensors	information about other objects

1.1.5 Resources and Collection

There are 2 resources, mass and energy. Mass is collected by carefully approaching and colliding with other asteroids. Energy can be passively obtained from cosmic radiation using solar panels or by converting mass into energy using generators.
If an asteroid collides with the base its relative speed has to be below a certain threshold. Otherwise it will shatter on impact and will not be collected.

1.1.6 Gameplay

The game starts with the base being in a stable orbit around the black hole and enough resources for initial orbit changes. Initially the player will have to use these resources to collect further resources by carefully colliding with other asteroids. When enough stuff is collected the base can be expanded or upgraded depending on the players needs. This necessitates the careful planing of how to spend the acquired materials. Will they be spend on maneuvering or base building?
Once the game has progressed far enough the play area begins to shrink which increases the demand for better and more efficient propulsion. If the player leaves the save zone the base takes damage and is eventually destroyed, ending the game. The player has to collect enough resources to build the necessary modules and stay long enough in the safe zone to compute the black hole entry trajectory. Once this is finished the base has to be deliberately steered into the black hole.
The game should be balanced and timed in such a way that it is finished (lost or won) in one session, so no save games are required.

1.1.7 Narration

Initially the gameplay itself will tell the story of surviving till the end.
If time allows, we plan to expand our story of escaping the collapse of the universe with narrative events during the game.

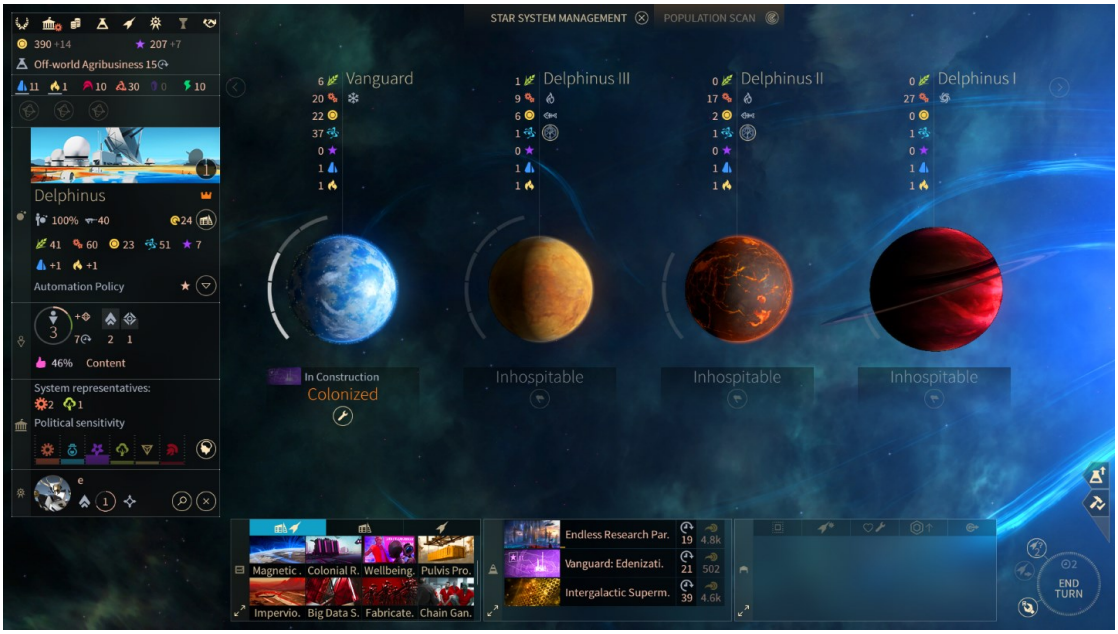

Figure 2: How Endless Space handles planets. We might do the base building in a similar fashion.

1.1.8 HUD

The required UI elements are: a list (later grid) view for base building, displays for stored mass and energy and the remaining time and an indicator for the base's trajectory. Later we also need an extended interface for trajectory planning.

1.1.9 Menus

In its early form the game does not need any menu-screens apart from the UI necessary to play it. If time allows it we might add a pause function and a main menu later on.

1.1.10 Graphics

At first we will have simple glowing wireframe graphics with some post processing for bloom (see Fig. 3). Later we will switch to solid flat shading with proper lighting (see Fig. 4). The black hole will be visualized by particle effects as seen in Fig. 5 once a particle system is implemented.

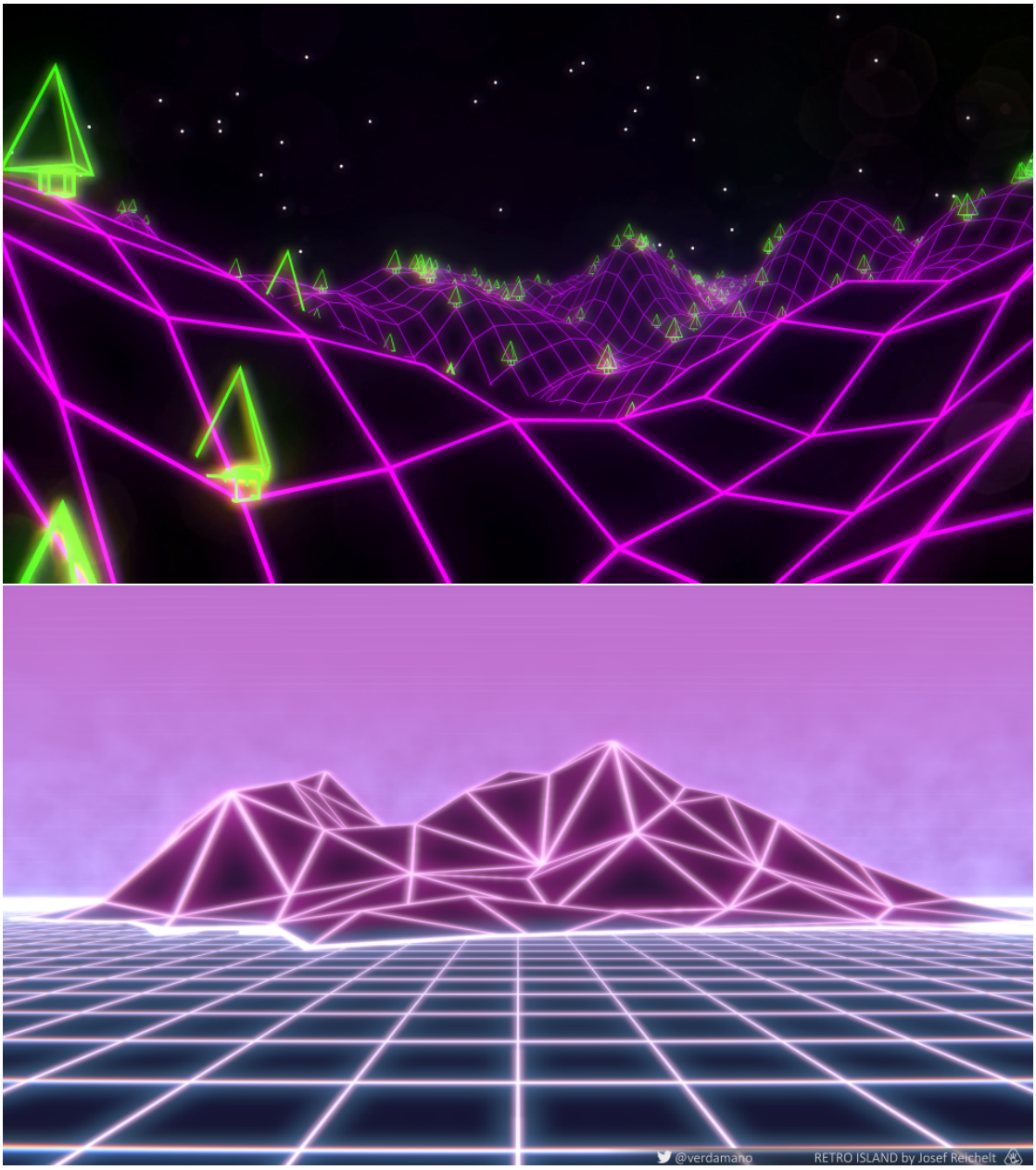

Figure 3: How the neon wire frame rendering might look

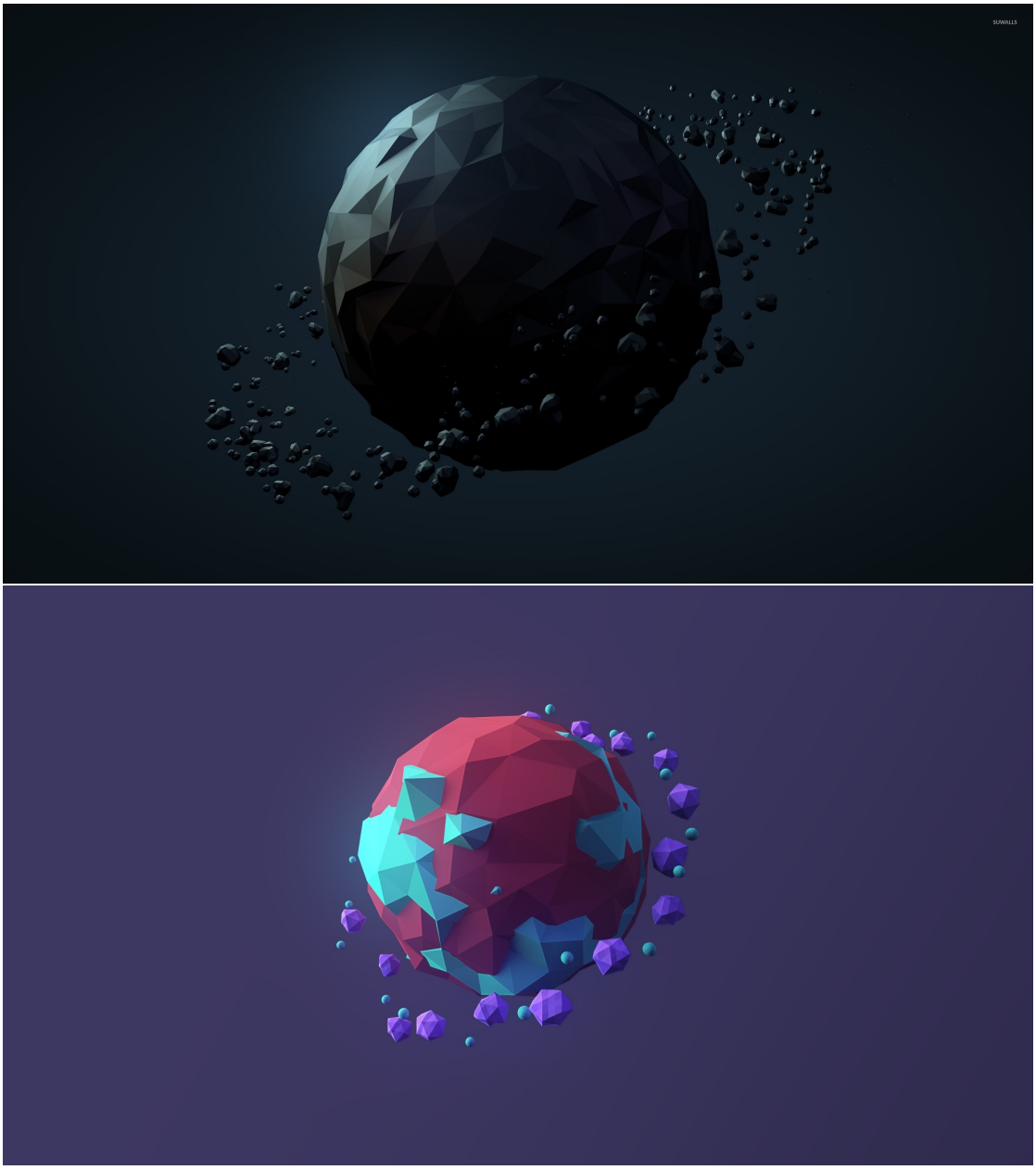

Figure 4: How the flat shading might look

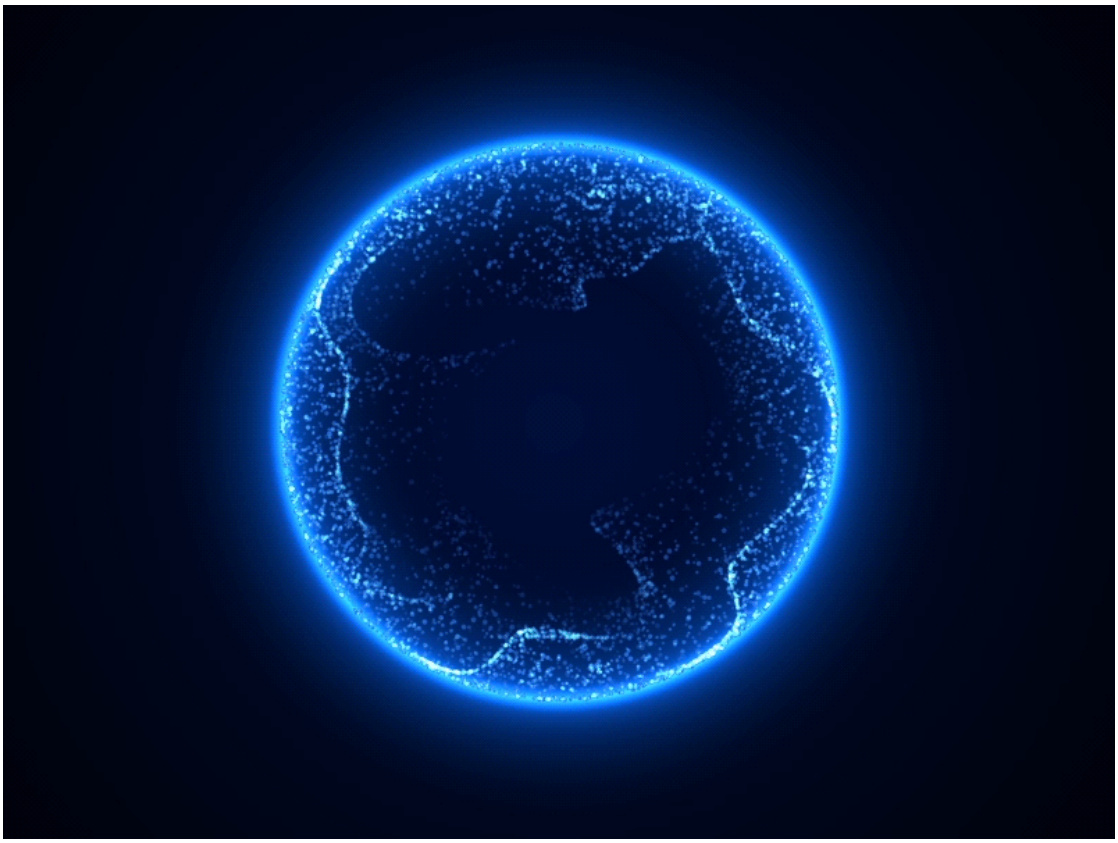

Figure 5: Black hole idea

1.2 Technical Achievement

Our technical focus is on the implementation of our own engine using C++ and DirectX12. Furthermore we aim to implement a working orbit simulation for all game objects.

1.2.1 Engine

The Engine will be modular and divided into several sub components to allows easy swapping of components. It will contain separate render and physics engines and the entity component system. This entitycomponent system will be similar to what Unity has to offer.
Initially the render engine will only provide basic wire frame rendering to speed up initial development because we don't need a proper lighting system. Our desired goal for rendering is a simple flat shading model with simple point lighting. This also includes a self written particle system to visualize the black hole.

1.2.2 Physics simulation

The physics engine will be able to simulate the orbits of several objects around a central body. It will only include the interaction of the orbiting objects with the black hole and not between them. The orbit oszillations of the black hole will also be ignored. All orbits will also stay strictly planar.
Initially Newtonian physics will be used to compute the orbit paths. All orbits will change if thrust is applied in any direction. This allows for objects to bump into each other changing their orbits and for the players to control their trajectory using thrusters.

1.3 "Big Idea" Bullseye

Core idea: orbit mechanics, technical feature: own game engine. See Fig. 6.

1.4 Assessment

The game will be played in short sessions of about 30-45 minutes. These sessions will be long enough to complete the game. Due to its nature as base building game its pace will be rather slow not unlike the Anno series. These short sessions also allow for a certain casual character since you don't need a huge time investment to finish a playthrough.
The game itself is characterized by its simple but stylish graphics, its planning and strategy aspects and the learning of orbit mechanics.

Figure 6: "Big Idea" Bullseye

1.5 Development Schedule

1. Functional Minimum This version will include the very basic version off the orbit mechanics. The game will also only include the resource mass which is used as fuel. The physics engine will only support sphere collisions. The render engine will only provide basic wire frame rendering.
   The story will only include the collection off the lost spaceship parts.

2. Low Target This version will include the first iteration of the base building feature (only lists).
   The resource energy for powering buildings will be implemented as well. This requires the addition of a resource management system.
   The render engine will also be upgraded to include a UI system.

3. Desirable Target Here the render engine will receive a major upgrade to include the ability to load 3D objects from files and render them using flat shading.
   The story will also be changed to include the computation of a decent trajectory into the black hole. This necessitates the addition of related science buildings to the base building.

4. High Target Random events will have their debut in this version. They will include things like visiting aliens, gravity changes in the black hole, or random asteroid storms. Defensive counter measures will also be added.
   A Kerbal Space Program like orbit visualization will also be included here.

5. Extras Time dilation effects might be a cool feature. This could have the effect of objects on lower orbits "speeding up" compared to objects on higher orbits.

   The ability to dock additional asteroids to your base to increase its size is a really cool idea as well but might be to expansive to include in our release version.

   The biggest improvement we could make is the implementation of truly three dimensional bases, similar to planetary annihilation.

2 Prototype

2.1 Prototyping Goals

We used the prototype to further condense our actual game play. We thought about the types of buildings we would need, what each one should cost and which types of resources each would need/ produce. Additionally we tried to improve the actual game play loop.

2.2 Setup

The very first prototype was created on paper. We drew circles in the middle of the page to represent our orbits. We further subdivided each orbit into several spaces. Initially these spaces on each orbit were calculated using the correct orbit formulas (so four spaces on the lowest orbit, 6 on the next, then 7, etc.). We ditched this idea since it proved to be to difficult to see where you would end up after an orbit transfer. Instead we opted to simply double the spaces in each orbit compared to the lower one. This made it very simple to see where you and everything else would end up after a movement phase. The downside is, that the outer orbits are very crowded with spaces.
We used simple dice to represent the asteroids. Resources, energy and buildings were simply noted on paper. You can see the final and more refined prototype in Fig. 7.

2.3 Game Loop

Our game loop consists out of five stages. These are movement, building, energy spending, orbit transfers and research.
In the movement phase all asteroids and the player station are move one space along their orbital track.
The next phase can be used to spend the acquired resources on buildings. This is entirely optional and does not have to be performed each turn. Used resources are subtracted from the resource pool immediately.
Next the player can chose to spend some or all of his energy on movement or research.
If the player choose to spend energy on research the acquired research is added to the research pool next.
Finally orbit transfers are performed. If the player lands on a field already inhabited by an asteroid, the asteroids resources are added to the players resource pool.

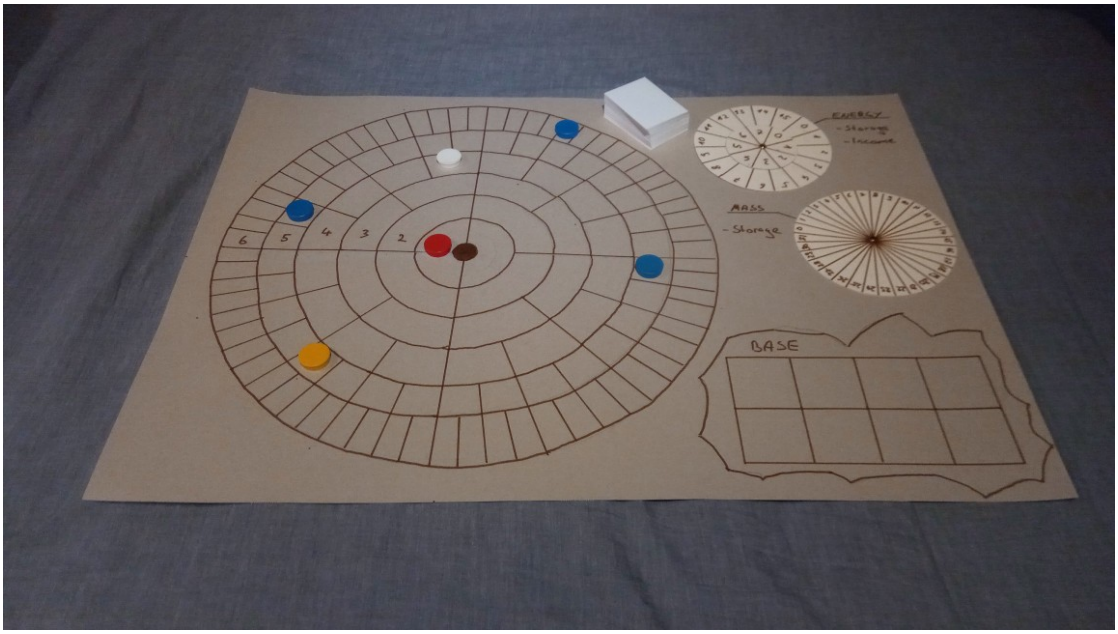

Figure 7: The paper game prototype.

2.4 Game Rules

2.4.1 General rules

The game is limited to a certain number of movement phases. In our current iteration we limited this number to be 100. In this time the player has to collect 200 research points to win the game.

2.4.2 Movement phase

Normally each asteroid on the board is only moved one space further anti clock wise. If this results in a long and tedious catch up process for the player, all pieces can also be moved by the same number of spaces. This number has to be subtracted from the end time counter.

2.4.3 Building phase

Each turn the player can build one building on an empty building spot. It is also possible to free one spot per turn by destroying the building on it.
A building can only be built if the required resources are in storage. After the building is build the used resources are immediately removed from storage.

Building	Cost	Effect
Engine	5 Mass	Multiplyer to Mass used
Generator	5 Mass	+1 Energy per turn
Battery	3 Mass	+3 Energy storage cap
Research	10 Mass	+1 Research -1 Energy per turn if used

2.4.4 Energy phase

Each turn the player gets new energy. He gets one energy base production and an additional one per built generator.
Energy is used to perform orbit transfers and power buildings.

2.4.5 Orbital transfer

If the player performs an orbital transfer he always lands in the next space in orbit direction in the orbit he moves to. If that space is already occupied by an asteroid its resources are immediately added to storage, the asteroid is remove from the board and a new one is spawned.

Size	Mass
small	3
medium	5
big	8

2.4.6 Random Cards

Every 10 moves one of the Random Cards is drawn and executed. The effect of these cards can be permanent, instant or a fixed amount of moves.

2.4.7 Physics

The actual orbit dynamics are simplified in the paper prototype. This is caused by the inability to properly display and simulate elliptical orbits. The game board would be way to cluttered with the possibility of eliptical orbits.

2.5 Results

The most obvious result was that out game is no fun as a single player board game. Having to move all the pieces each turn gets annoying rather quickly. Orbit transfers are also not that interesting in the board game version since they require not that much amount of planning.
On the upside seeing objects on the outer orbits slowly crawl along while you zip

across the sky on the inner ones is quite fun.
We also realized that the player got way to much resources far to early/ easy. This removed a lot of tension since you could simply throw them around without any care in the world. This also made the base building kind of useless since you could build everything you wanted at any time. Thus completely removing the need for proper planing.
We further realized that the lategame would be a waiting game until reaching the amount of reasearch neccessary to win, after the base ist build properly. The introduction of random events solves this problem.

3 Interim Report

3.1 Implementation

We subdivided our engine into multiple loosely interconnected parts:

- Render Engine
- Physics Engine
- Input Engine
- Components
- GameObjects
- GameMain

GameMain is the Main Class of the Engine wich connects all parts of the Engine and implements the game loop. As the name suggests the render engine is responsible for rendering everything from 3D objects to the UI. The physics engine computes collisions, calculates/ advances orbits and will handle all further physics related tasks. User input is captured by the input engine. The entity component system (With GameObjects as entities) manages game objects and provides functionality to them. It es essentially the scene graph for our engine. Calling the individual engines, keeping the frame rate stable and managing the game in general is done by the GameMain class.

3.2 Physics engine

Right now the physics engine handles collisions and orbit calculations. If any further physics related tasks pop up in the future they will be implemented in this part as well.

3.2.1 Collision

Right now our engine only supports sphere sphere collisions. Initially this will be completely adequate since all objects in game can be reasonably well approximated with spheres. There are also no optimization techniques like kd-trees right now, so all physics objects are simply tested for collisions against each other object.

3.2.2 Orbit mechanics

Orbits are simulated using ellipses. Initially i thought about simulating them using the gravity equation. But i know from experience that this approach leads to instabilities rather quickly. Therefore the current implementation uses ellipses, the orbital elements (eccentricity, inclination, etc) and geometry to compute the orbits. This also comes with the added benefit of making time warp (like kerbal space program can do it) really easy.

3.3 Entity Component System

The system is heavily influenced by how Unity3d handles game objects. Everything the player sees or interacts with in our engine is either a game object or a component. Game objects are container objects that manage all components needed for this object, thing, etc. Components then provide the actual functionality. Let's use an asteroid as an example. The asteroid would be one game object. But the game object itself does not do much. So we attach a render component to it so that it can be seen on screen. If we want the asteroid to be affected by collisions we attach a sphere collider. The orbital behavior is handled by an orbit behavior component.

As in Unity we also provide a behavior class. It works similar to Unity's monobehaviors. If new game logic needs to be added to the game you can simply create a new class which extends behavior and attach it as a component to a game object. The behavior component acts as base class for all generic components for better differentiation with engine-used components.

The game object class is responsible for managing all its components. So if new ones are added they have to register themselves at their game objects which in turn handles them appropriately. This includes informing the respective engines. Game objects also handle their own and their children's deletion. This created an interesting problem of what to do when they should be deleted. It can happen that some code decides that a game object should be deleted. But other parts like the render or physics engines might still need it. Therefore objects are only flagged for deletion and are then actually deleted at the end of each frame once everything else has finished.

Every game object comes with an already attached Transform component. It represents the backbone of our Scenegraph and manages all positional information and functions. The already implemented version is build with performance in mind and updates only changed and needed matrices. Internally quaternions are used

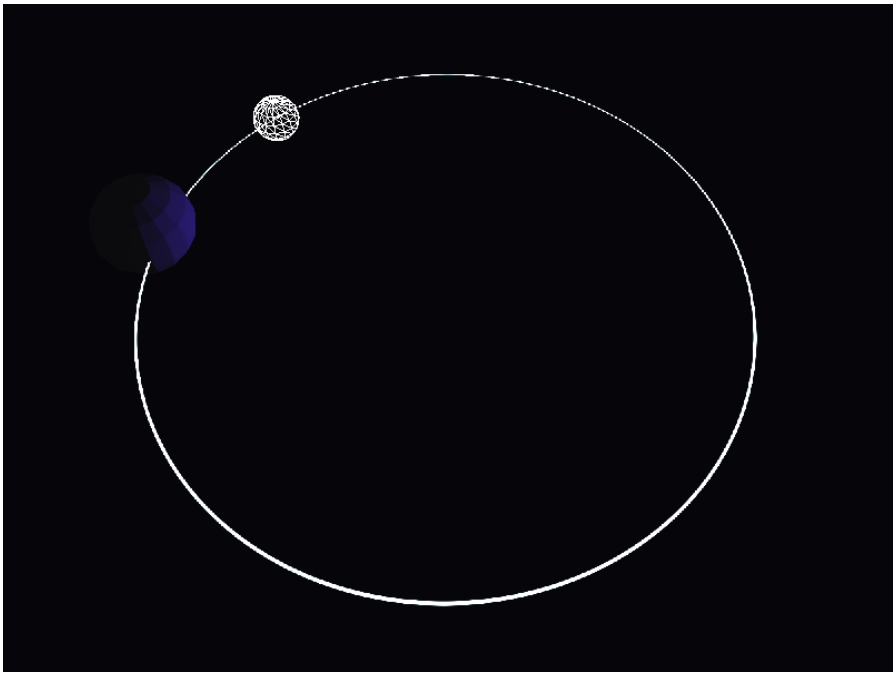

Figure 8: Wireframe and Flat shading and orbit rendering.

for best performance and usability (eg no gimbal lock).

3.4 Render Engine

The Render Engine manages the rendering of objects as well as all necessary resources and their descriptors. It is implemented in the DirectX12 API. The render pipeline is based on the existing renderer in the template project we use. It is extended and heavily modified based on various tutorials and other sample implementations.
Currently the Render Engine supports rendering of RenderComponents classified opaque, transparent or UI. It also includes a post processing stack. It includes texture resource creation from binary data (no file loading yet, also likely not needed) and the generation of mesh resources (vertex and index buffer). Meshes can be either static meshes (cube, plane) or generated meshes (sphere with variable tessellation).

3.4.1 Object Rendering

Wireframe rendering (see Fig. 8) proved easier than expected, as it is just a single flag for the render pipeline. Flat shading requires constant normals for each triangle, but the current mesh rendering uses vertex indexing, allowing for only

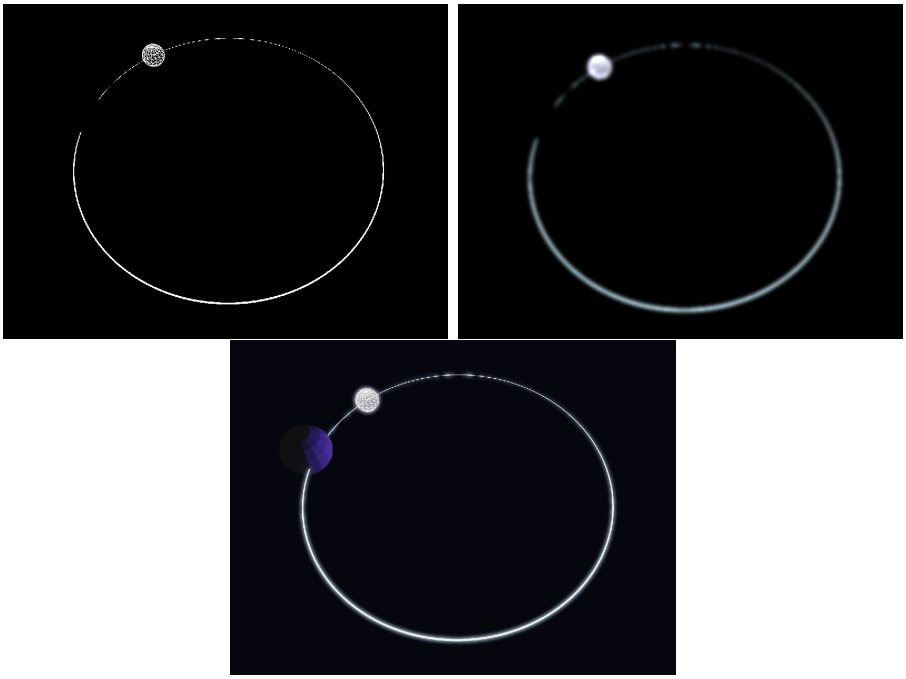

Figure 9: Bloom: filter to get only the bright parts of the image (top-left), blurr (top-right) and blended onto the final image (bottom, affected by tonemapping).

one normal per vertex. Instead of switching to non-indexed meshes, the normals for flat shading are simply generated online in a geometry shader for each triangle primitive. Lighting is not yet implemented, apart from a hard-coded central point light.

3.4.2 Postprocessing

Postprocessing needs additional render targets that can also be used as shader resources, so the first step here was the setup of additional texture resources with their render target and shader resource views. As all effects operate only on already rendered scenes they only differ in the pixel shader.

The first implemented effect is a simple bloom effect (Fig. 9), consisting of a filter pass to get the parts of the image bright enough to be affected by the bloom effect and a bilateral blurr pass, run multiple times for variable bloom strength. Second, somewhat needed by the bloom effect, is tonemapping, based on a sample implementation, to allow for HDR color remapping to SDR range [0,1]. The tonemapping shader also blends the result of the bloom pass with variable intensity. Lastly I added SMAA (Subpixel Morphological Antialiasing), by simply porting the original implementation to DX12 and shader model 5.1. Currently only the

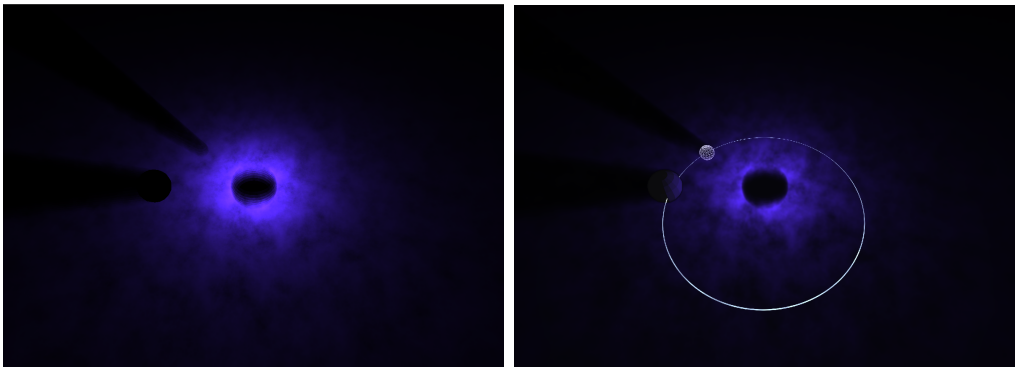

Figure 10: The raymarched fog alone (left) and blended onto the final image (right)

SMAA 1x version is working and active. See Fig. 11 left, the effect however is barely noticeable.

Extras: Somehow I wanted to implement volumetric raymarched fog, lit by a central light (black hole) with objects casting shadows into the fog. I used a bachelor thesis explaining a implementation for Unity for directional light and lecture slides. The central shadow casting point light made a shadow (depth) render pass and a system to render cubemaps necessary. Currently this is implemented as six render passes to six different render targets with matching view matrices for the six sides of the cube. The actual raymarching shader then uses these depth maps (to determine light occlusion) as well as the viewport depth (view occlusion) to render the fog. The fog density has its maximum in the x-y plane (our orbit plane) and decreases linearly with distance. It is also affected by multi fractal noise. The result of the raymarching is then blurred and a blended on the main image (see Fig. 10).

The render pipeline, with all techniques active, works as follows. First the depth cube-map is rendered for all opaque objects. Then all opaque objects are rendered (color, normal and depth) with the normal viewport, followed by transparent objects (only color). The Fog is rendered and blended on the main texture before the bloom effect is applied. The result of the bloom pass is blended on the main texture before the tonemapping. The last technique is SMAA, followed by UI objects, which should not be affected by image effects.

3.4.3 UI

The visible UI is currently only a text renderer. The necessary texture for the font lead to the creation of the texture resource creation system. On creating

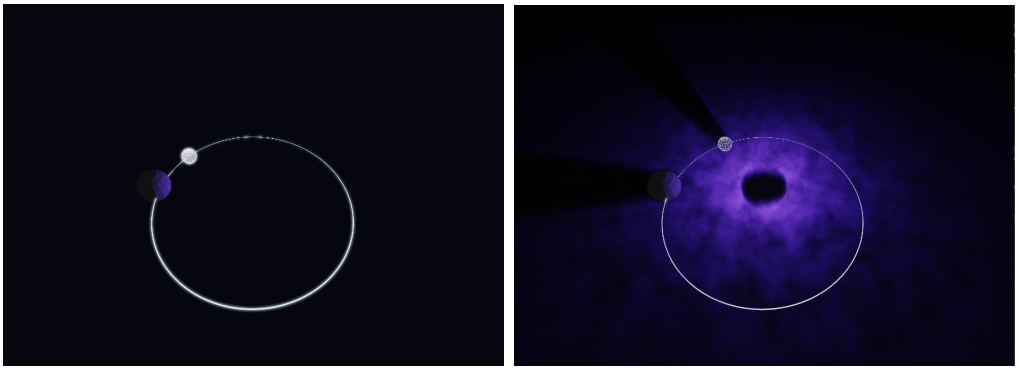

Figure 11: SMAA (left) after bloom, compare to Fig. 9 bottom. Right is the final result with all post processing applied. Compare to Fig. 10 right to see the effect of tonemapping.

a UIComponent, character and vertex buffers are allocate for a maximum string length. The string is then parsed to create a per-instance vertex buffer with each vertex holding the information of one character (position, glyph). For each instance four vertices are rendered, generating a quad with correct position, size and texture coordinates in the vertex shader. Finally the SDF (signed distance field) font texture is sampled and evaluated in the pixel shader.

3.5 InputEngine

The implementation of a InputEngine was harder than expected, mainly because of the UWP Template we used. All Window functions, including the Eventsystem, are wrapped into a CoreWindow Class. This class is a C++/CLI Class and required special syntax and structure we did not knew from normal C++. The InputEngine tries to make the use of Userinput easier. It offers captured User-Input on a per frame basis for Behaviour Components and the possibility to register rectangles in sreenspace with callbacks for realtime eventhandling for Userinterfaces.

4 Alpha Report

4.1 Gameplay

4.1.1 Orbit mechanics

The orbit mechanics were finally put into a stable form. This took much much longer than initially anticipated. The version from the last milestone suffered from presumably random jumps when the player tried to change the orbit. These where caused by several issues.
The conversion from orbit elements to orbit state vectors was wrong. Using a different formula to compute them fixed this issue.
The other main issues where caused by the acos and atan functions. Both only return vectors in the first and second quadrant of the unit cycle. So whenever an orbit was in one of the other two remaining quadrants it would flip back to the first two. Finding and fixing all of these computations took quite a lot of time.

4.1.2 Resources

The game has two main resources. Mass and energy. Both are needed in varying quantities to construct buildings. Mass is also used to power your engines.
Mass is aquired by collecting asteroids. Energy is produced by reactors.

4.1.3 Base building

This milestone also saw the initial version for the base building system. A very basic version of it was realized using only buttons with text.
A number of building slots was added on the bottom of the screen. Once you click one of them a new menu showing the available buildings opens. You can then select one of them to build it.

4.1.4 Asteroids

Asteroids are now spawned on random orbits, their mass can be collected on collision.

4.2 Renderer

Apart from the star background there are no obvious changes of additions to the render engine since the Interim milestone. There are quite a few internal changes and improvements as well as minor visual improvements.

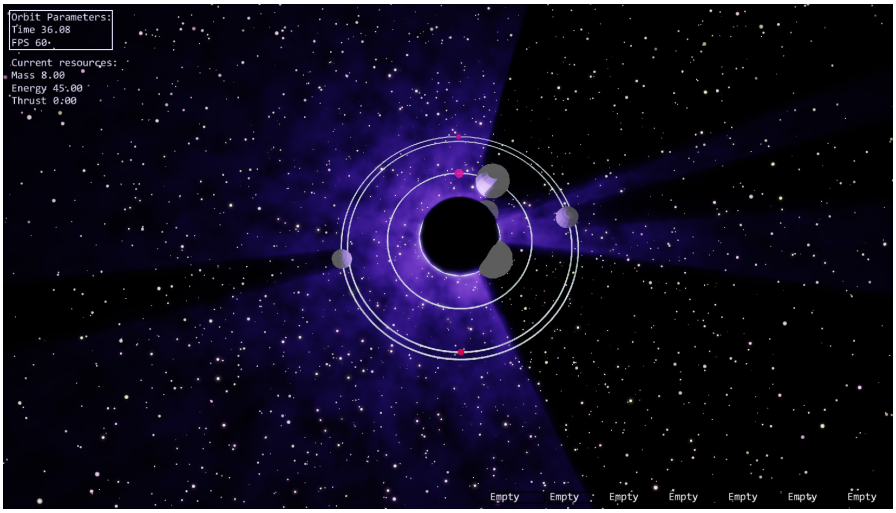

Figure 12: The current final render result.

4.2.1 Additions

Camera. A camera component has been added. This is a simple component that is registered with the render engine and forwards the transform of its parent GameObject as view matrix in order to be able to move the view by moving a GameObject. A camera controller for panning and zooming has also been added.

Bilateral blur. In addition to the normal Gaussian blur (used for the Bloom effect) a bilateral Gaussian blur, using the depth texture, has been implemented. It is used to blur the rendered volumetric fog to avoid halos around objects.

Particles. A system for rendering sprite particles has been implemented. This includes a material for particles and a ParticleRenderComponent. The render order of particles is not sorted. They are rendered after the transparency pass with additive alpha blending: $dst = dst + \alpha * src$ and do not write to the depth buffer to not occlude other particles.
As no particle system has been implemented yet the particle rendering is only used for the background stars. Particle positions are randomly generated in the vertex shader using a simple Xorshift RNG that is seeded with the vertexID (i.e. the index of the particle). (I took the RNG inplementation from one of my previous projects). The particle shaders and elements of the render engine can later be used to render particles from a data buffer provided by a particle system.

4.2.2 Improvements

UI. The parameterization of the UI render components has been improved for consistency across different viewport resolutions. A UI element now has a relative anchor position, given in screen-space UV coordinates, an absolute offset and a size in pixel. This ensure that UI elements are not skewed or move out of the window when resizing.

Cube map rendering. The cube map rendering has been refactored from 6 separate render passes (to 6 separate textures for the 6 sides) to a single render pass to a texture2DArray with 6 elements. This approach uses a geometry shader to generate the primitives as seen from each side with the corresponding view matrices. Instanced rendering with 6 instances per object could also be used.
The shadow cubemap is now also passed to the shader for flat shaded meshes, such that they now receive shadows from the central light as well.

The material class has been expanded to support more blending types: opague, transparend, transparent additive (for particles) as well as write or discard for the normal-depth texture.
Using the new material blending types an invisible occluder has been added to hide objects behind the black hole.

4.3 InputEngine

We wildly underestimated the time we would need to implement User Interactions. It was one of the last pieces left before we could start implementing gameplay and could not be postponed further.

4.3.1 Changes

As described in the previous section we had some problems with event handling. The problems were not solved because we needed a way of generating own events for our UI. A simple pass-trough of events is not possible because it is not allowed in C++/CLI to return unmanaged objects, including all containers from the standard library.

InputWrapper. We decided to rename the old InputEngine to InputWrapper, which is only responsible for giving an interface for all Events.

InputEngine. The new InputEngine processes all events recorded by the InputWrapper and generates new, usable Events on a per-frame basis (analogical to the other Engine parts we use).

4.3.2 Additions

InputEngine. The InputEngine now supports full interaction with our in-Engine UIElements. Our UIComponents register in GameMain, like all other Components, and will be processed once every frame by the InputEngine, which then calls the appropriate Event-Methods (OnClick, OnPointerEnter, OnPointerExit and OnPointerMove).

UIComponents. These are a new primary type of Components, representing Objects interacting with the User in screenspace. One instance of this new family is UIButtonComponent. The UIButtonComponent uses a combination of rectangles and text to visualize user interactions. The Component is highly reusable because it uses callback functions defined at initialization or anytime during runtime.

5 Playtesting

5.1 Organization

In general we recruited mostly friends and family. The few notable exceptions were guys from Phillips/ Moritzs tutorials. The players were then let loose upon the game without any explanations. If any questions arose or if players seemed to be completely stuck they received enough help to overcome their obstacle and nothing more.

5.1.1 Questions

After each play test each participant was given the following questions to answer:
-How do the controls feel?/ what do you think of the controls? -¿ could they be improved?
-What do you think of the overall gameplay?
-Did anything feel odd/ out of place/ not fitting the game?
-What do you think of the pace of the game?
-Did you understand the controls?
-Did you understand the HUD?
-Did you feel that something was missing?
-Something you would add or remove?
-Have you been bored at anytime during gameplay?
-Have you been confused/you didnt know what to do at any time during gameplay?
-Was the game difficult or easy?
-Did you have problems to achieve, what you wanted to during gameplay? -If so, what was the reason?
-How easy or hard was it to collide with the asteroids?
-Did you understand the orbit mechanics?
-How could the game be improved?
-Anything else you would like to mention?

5.1.2 Answers

Understanding the orbit mechanics was the core aspect of our game. As these are quite a hard subject to grasp immediately most players had trouble understanding them on their first try. But given a few attempts all of them understood them well enough to play the game somewhat satisfactory. This was to be expected.
In general many players perceived the game as to slow paced. This might be mitigated by the time warp but so far seems to be an inherent property of our game.

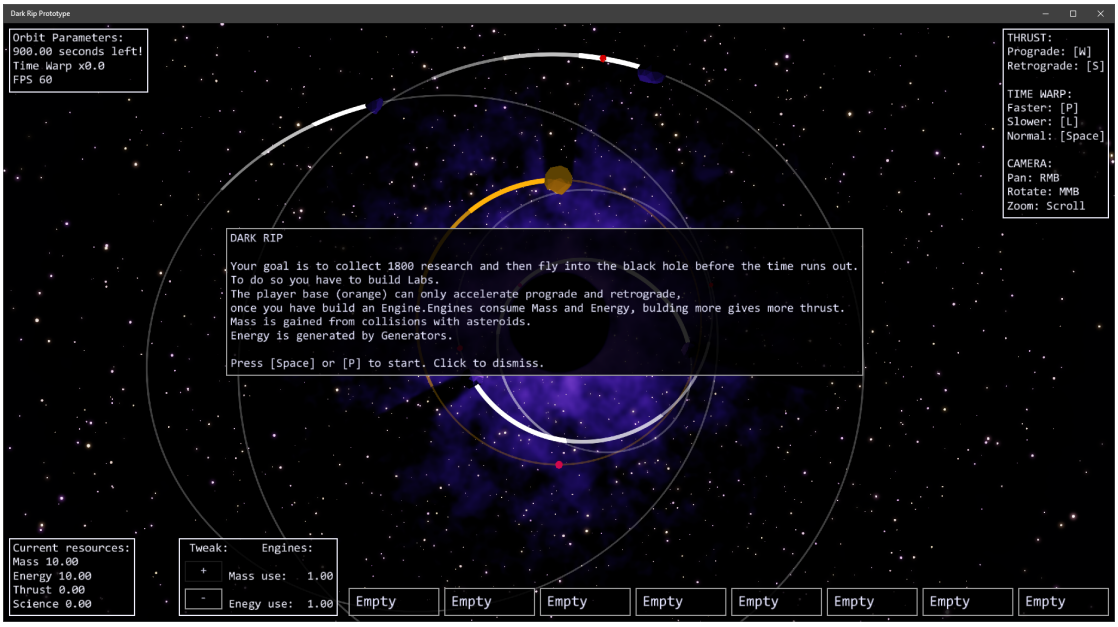

Figure 13: The current state of the game

The UI seems to be rather well designed as most player understood it quite easily. Only the building slots were not immediately found. The same can not be said for the controls. Most players initially struggled with them. In general most players remarked the lack of (cool) feedback (explosions on collisions, sound, colored HUD, etc.).
One of the reoccurring complaints was the initial lack of any buildings coupled with the low start mass. Most players found this somewhat counter intuitive. How did the player get where he is without an engine. One player remarked that he did not get enough mass.

In general most players seemed to like the game, but found it rather unpolished.

5.1.3 Changes

The most basic change we implemented was increasing the thrust of engines to combat the initially slow game play. Additionally we added a short tutorial text at the start of the game. We also increased the building slot size to improve their visibility. But the most important change of them all: BUG FIXES!!!

6 Release

6.1 Changes since Playtesting

Engine Structure The orbit mechanics were originally implemented in the physics engine. Although this worked, it was not really a task for the engine since they are very game specific. To facilitate reuse of the engine, all computations related to orbit mechanics were moved to a separate component that manages them.
Additionally we fixed a bug where the deconstructors of components would not be called upon deletion.

Balancing In the beginning many players struggled quite a lot with the game since they either ran out of resources or time. To combat this we increased the start mass a little bit to allow for more maneuvers at the start of the game before any asteroids are collected. We also increased the total time to give players more room for understanding the orbit mechanics.

Particle System We implemented a GPU-particle system where the particle data reside exclusively in GPU memory. compute shaders are then used for particle spawning, update and state keeping. The rendering then uses the same per-particle data buffer. Since it would be infeasible to sort the particles, we use order-independent rendering of particles: alpha-additive blending, depth test and no depth write (this is a simple setup in the blend description). The fog however respects particle depth, using a secondary depth buffer.

6.2 State of the Game / What We Achieved

Our core goal of building our own engine from scratch was completely archived. We managed to implement all the necessary parts we needed for our game. The engines core is an entity-component system similar to that in the Unity engine, with GameObjects and Components as well as the different engines for rendering and physics.
Our physics engine only supports sphere sphere collisions as originally planed.
The render engine supports meshes loaded from obj-files, materials, real-time point-lights, screen overly text, volumetric fog, a GPU-particle system and post-processing steps which include bloom, SMAA and tonemapping. It is also sorts objects into different rendering passes.
The UI is composed of the InputEngine, which is responsible for detecting mouse and keyboard inputs and handles clickable on-screen buttons, and a UI-button system.

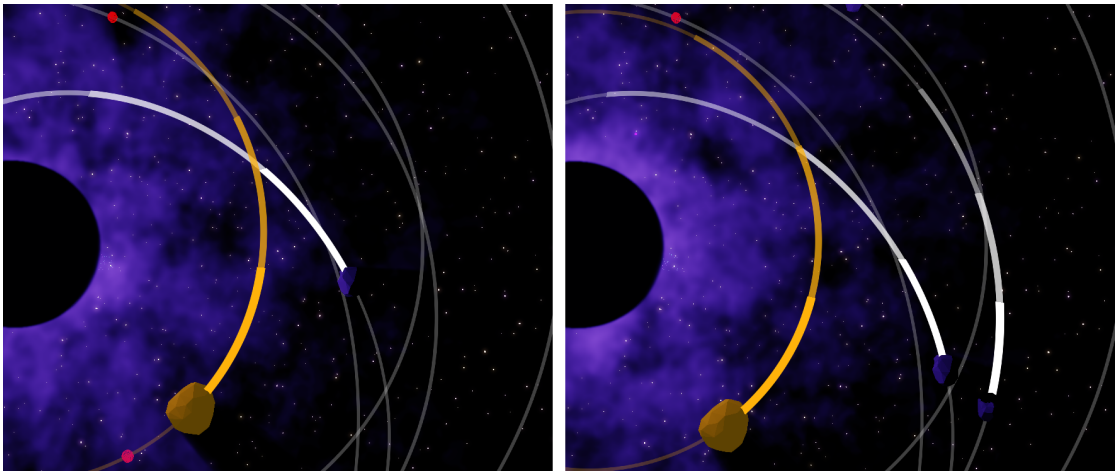

Figure 14: Left: The indicators don't align, therefore there won't be a collision. Right: The indicators align perfectly at the ten second mark, there will be a collision.

The actual game itself supports physically accurate orbit mechanics (two-body simulation only). Furthermore we managed to complete the base-building basics and the resource system. We introduced a simple marker to help recognizing orbit intersections. It shows the position of a body five, ten and 15 seconds in the future, indicating a collision if the markers of the same time step are close enough together (as shown in Fig. 14).

We did not manage to implement our high target goals, apart form rendering related tasks. This would have included improved base building, random events and a defense system. Additionally we had planned to implement an orbit intersection planning tool, which was only partially achieved.

A representative screenshot of our final Game can be seen in Fig. 15.

6.3 Evaluation

We managed to keep to the original schedule until the weeks around Christmas, where we began to lag a bit behind. This was further amplified by blockers such as the orbit mechanics or the user input.
We compensated these delays by temporarily excluding our high target and stretching our remaining tasks over the rest of our time frame.
However, Erik added some visual features that were not (explicitly) planned, but drastically improved the visual quality of our game.

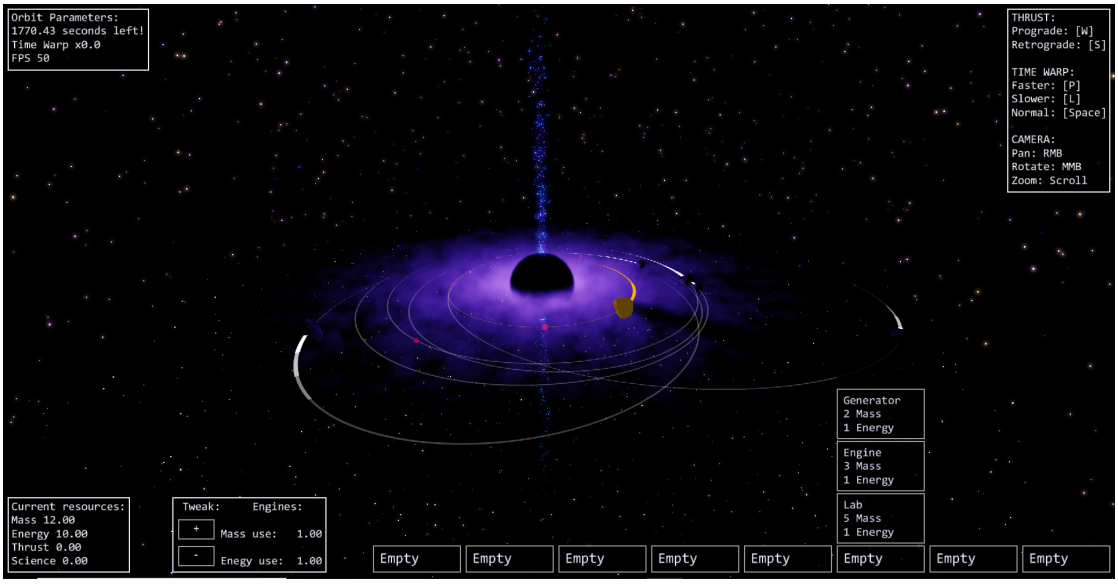

Figure 15: Final visual of our game, including the new GPU particles.

6.4 Personal Impressions

1. What was the biggest technical difficulty during the project?
2. What was your impression of working with the theme?
3. Do you think the theme enhanced your game, or would you have been happier with total freedom?
4. What would you do differently in your next game project?
5. What was your greatest success during the project?
6. Are you happy with the final result of your project?
7. Do you consider the project a success?
8. To what extend did you meet your project plan and milestones (not at all, partly, mostly, always)?
9. What improvements would you suggest for the course organization?

Moritz

1. The biggest struggle on my part was to make the InputEngine work with the UWP (Universal Windows Platform) Project Template.

2. "Twister" was a smart choice for a theme, because it can be interpreted literally or very abstract. But i think it lead most of the time just to something spinning in the game.

3. I think that some restrictions can help creativity and it really helped in the

first phase to get ideas and develop them in a limited space.

4. I wouldn't use any template as codebase, especially not the UWP-Template. It was way more complex to make later additions work rather than to implement a window with a message pump ourselves.

5. The Transform class (responsible for all positional data and calculations) broke not once and had no problems.

6. It would be nice if we had a bit more gameplay (high target).

7. I think we archived everything we desired.

8. We hit every milestone as we wanted, but our original timeline was a bit more compressed and we initially planned 2 additional weeks for high traget tasks and 1 week for final bugfixing. But planning these as buffer turned out to be a good choice.

9. Make the first phases a bit shorter and add some time after playtesting.

Phillip

1. For me the most challenging part was getting the orbit mechanics to work properly. The initial orbit movements were rather easy. But the ability to actually change your orbit proved quite challenging. Most of this was down to solving the related equations. Once this was done i spend a really long time to handle all the corner cases.

2. I really really liked the theme. It was open enough to allow for a lot of cool interpretations but restrictive enough so we would not get lost in ideas.

3. The theme added focus in the design phase.

4. Spend more time on the initial schedule to identify all necessary tasks.

5. Implementing the entity-component system. In my opinion it is the backbone of the entire engine. So getting it to work so smoothly really sped up the game implementation time in the end.

6. Yes. Although the game itself could still use a bit of love, implementing our own engine was the main goal. And we archived that completely.

7. Again, yes.

8. Overall we are somewhere in the high target. But that is mostly down to Eriks work on the graphics. I personally finished all my tasks from the desirable target.

9. I would like the ability to focus on a purely technical aspect instead of writing a whole game. It would have been nice to focus solely on the engine.

Erik

I worked mostly on the render engine and everything DirectX related and enjoyed that very much. I also learned a lot, more than I think I would have had we used an engine like Unity (again).

1. There was no really big technical difficulty for me, the most time consuming problems were these nasty small bugs where you keep overlooking some wrong sign/formula. I think the most difficult part was making the text rendering behave the way I wanted it.

2. It was there, to be considered during the design phase. After that I didn't really think about it anymore.

3. It is nice to have some guidance, as long as it does not restrict the creative process. Maybe give 2 or 3 themes or use it just as a suggestion/inspiration, although I think the theme was already abstract enough for that.

4. I would not not use an UWP-app template again. Also, now that I have some experience with DirectX 12, I would plan the Render engine better and more detailed.

5. Towards the end where I got the impression I understand enough of DirectX 12 and could just implement stuff (GPU-particles) without having to look up everything.

6. Quite happy, yes. In the end I implemented all the graphics I wanted. The volumetric fog could look better from up close and its performance needs to be improved, but performance was not an important aspect for a project of this size. A bit more gameplay would be nice.

7. Overall I consider the project a success. We achieved most of our initial goals as well as some extra rendering/graphics stuff I just wanted to include.

8. Again, mostly. We got all of the desirable target, missing moist of the high target, but I also added stuff that was not planned (Fog) or in the extras.

9. More time and larger groups would be nice.