Interims Report

1. Functional Minimum

Tsunami Game Engine

The engine needs to support gameobjects with basic components attached to them. For these, a scene structure was needed to integrate the objects in a controllable way. Resource loaders are an important element for the engine to use memory efficiently, taking care of managing the application resources. To combine our technical challenges, a render framework and physics framework are the core features of our engine.

Graphics

Raytracer

The first rendering pipeline available in the engine was raytracing. To achieve this functionality we created a volumetric material on the engine side that holds on to a volume texture (our water representation) and some additional data like volume extent. In this stage we use a single compute shader. For each pixel, we do ray/box intersection test using the slab method and, if we detect a hit, we use the SDF contained to interpolate the surface position and compute light contribution, we keep on marching on the ray to accumulate water depth.

As our rendering uses an hybrid approach, we also need to create a depth buffer for later use. Render targets are read/write textures created with multiple views (UAV and SRV) for later use in the rendering loop.

Rasterization based Pipeline

This stage collects the *Solid Render Queue* from the scene, this is a collection of all the objects using triangular meshes and solid material. At the time of this writing, the engine does not support depth sorting or frustum culling. This decision comes from the fact that our entire scene will be contained in a pretty much static view (render camera will only be doing small rotations) of the world. Once the objects are gathered and the PerFrame data set in the pipeline, we loop over the objects, one by one they set their own PerObject material, make their Draw call and reset if they own shared memory for later use.

As default shading technique we use Blinn-Phong light evaluation.

This stage outputs a color target and a depth target used later in the rendering loop.

Physics

Liquid Simulation

For the liquid simulation we used a precomputed FLIP simulation, computed with the framework Mantaflow. We tried to reconstruct the wave as realistic as possible using a similar underground shape and the bridge pillars on the inflow. We saved the final simulation as 4D SDF grid, an implicit form where each cell contains the signed distance to the surface of the water. In the game we load the simulation as is and loop over the 4D grid, extracting a 3D SDF grid which will then be rendered with the raytracer, but will also be used for the buoyancy simulation and the particle effects in a later step. We use an implicit representation because it makes it a lot easier to simulate the buoyancy and the flow forces. It gives us also the possibility to insert advanced effects with the raytracer.

Rigidbody Physics
As we implemented the whole game from scratch, we also implemented the rigidbody simulation on ourself. We started with really simple functionalities which includes forces acting only on the center of mass and some simple integration techniques for the velocity and position update. To keep it simple we worked without any rotational components.

Water-Object-Interaction
The water-object interaction is one of the most important parts of the physics for the game. The whole game will base on how the surfboard behaves on the wave respecting to the buoyancy and the flow forces. We wanted to make it as realistic as possible, therefore we used like mentioned before the SDF representation of the water. Thanks to this we can easily check if a specific point is underwater or in air, making a simple lookup in the 3D SDF grid. Not only this, we get also the depth of the point like for free.
For the required minimum we worked with a sphere which has allowed us to simplify the buoyancy calculation. We used a really brute-force approach by checking each cell of the grid if the distance of the cell to the center of the sphere is more or less like the radius, if it was we calculated the local forces for this cell. The sum of the local forces gave us then the required forces.

Input
Basic Input Scheme
The input consists of the simple input with a keyboard. The standard message queue of Windows is used to identify certain generated input messages coming from devices. *WM_KEYDOWN* and *WM_KEYUP* are message types informing the application that certain messages are generated by keyboard input. These can be used to check for the virtual keycode inside the message to find out which key has been pressed. This information is used to fill a state table of supported keys, which can be checked with a system call *GetKeyState(VK_CODE)*, if the key mapped to the virtual key code has been pressed in this frame.
As this input scheme is very simple and doesn’t allow for raw input of more devices of the same type or even getting more information regarding the respective device and the processed data, a more sophisticated method to handle input needs to be supported.

2. Low Target

Tsunami Game Engine

The game engine went through several iterations to become more efficient regarding memory usage and processing gameobjects in the scene. To combine opaque elements with transparent ones including a rasterizer rendering and raytracing, the engine allows for different pipelines being processed and merged to a final image. Also a Graphical User Interface system is integrated which will write text or images on the screen. Different aspects will be combined and used by the engine in scripts, which are attached to gameobjects in the scene and updated every frame. Therefore, custom operations can be combined in the update routines of one script.

Graphics

Rendering is realised in multiple render passes called *pipelines* inside the engine. A pipeline is a high level structure that contains shader programs, render targets and rasterizer information. To use a pipeline, the user must initialize and compile the different shader passes and simply *set* and *reset* states before and after drawing respectively.

Pipelines are not enough to visualize an object, a GameObject must have a RenderComponent attached in order to be drawn on the screen, which contain a Material and a Mesh. Different Materials can implement different render modes.

Materials are a core object in the rendering process, they hold on to the objects' specific data like color and textures. Based on the material type, the Scene creates *render queues* that will be used from the render loop to select which object must be drawn in which render pass. Materials have two key methods, *set* and *reset*, those are used to bind to the current active pipeline what we call *PerObject* resources.

Shaders also need some data which is common to every object that we must render in a render pass, we use the Scene to gather tagged objects to create *PerFrame* data that is used as a one time bind before rendering each pipeline.

The rendering loop currently is realised in five pipelines:

1. Raytracer
2. Solid
3. Merger
4. GUI
5. Font

1 and 2 are virtually parallel and they both output a color texture and a depth texture.

Output from 1 and 2 is used as input in 3, in a compute shader we sample each fragment depth from raytracer and solid and, if needed, we perform the blending between the 2 color textures. This pipeline outputs a color texture which is used in 4 as input. 4 gets a collection of GUIMaterials holding on to a collection of textures and their position in normalized device coordinates. Again a compute shader will perform fragment/texture intersection and performs the blending if necessary.

Fonts are rendered using DirectWrite and Direct2D APIs. To realise this feature we had to create a subsystem called FontEngine that will take care to hold on to required data structures used for font rendering. Again we use materials to create strings, text alignment and colors used while rendering different strings on the screen.

Physics

Rigidbody Physics

For the low target we added some functionality to the rigidbody implementation. We added rotational components and impulses for the collision handling. For the inertial momentum we approximated it with an inertia tensor calculated with the mesh of the object and his surface.

Water-Object Interaction

Also the water-object interaction was extended to handle arbitrary shapes.
The buoyancy and the flow forces are both forces caused by the pressure of the water. The pressure depends on the surface-area of the object and for the buoyancy also on the depth, for the flow forces on the difference of the water and the object velocity. To approximate the surface-area of the water, we saved for each vertex of the mesh its voronoii-area as a vector, which points in the direction of the normal. This can be precomputed and transformed in the needed form during runtime.
Our final algorithm checks first if objects are near the water using simple bounding volumes. The concerning objects will then be transformed into the local space of the water domain. For each object the vertices will be mapped to the SDF grid and the local buoyancy and flow forces will be evaluated for each vertex.
By using simple dot products between the local forces and the area vector the resulting force can be calculated. Also some additional forces like friction can be computed using a cross product between flow forces and the area vector. The single forces will be added up to a main force for the object. Also the points where the forces are acting will be averaged to get a single point, weighted by the magnitude of the forces. Finally we get a force which we will apply on the object at the given point.

Input

Robust Input Handling System

The input of the game is handled by the *RawInput* API of Windows. RawInput is generated when a top level collection of input devices regarding their usage page and usages has been registered. Otherwise the system will never receive raw data from devices.
After a device has been registered, the handle to a device will be known by the Input Manager, making it possible to support several devices of the same type. For debugging reasons, a mouse and keyboard needs to be supported, so the Raw Input scheme seems like a good idea.
The complete input handling system will be accessible by calls to the *InputManager*. This is the top level class handling every type of raw input. The *InputManager* passes the data received to other classes. The *DeviceManager* will register any new devices, receive the data and forwards that to the respective device. It will create and destroy devices whenever needed. It also maps the virtual key codes for keyboard input to application specific values, which are easier to read and use, so a raw to virtual mapping is done.

The last two intended classes, *InputMapper* and *ContextManager*, however, have been postponed for the remainder of the course as the RawInput took very long to integrate into the system. The documentation was unclear in some crucial places and certain combinations of data structures weren't allowed to be used for some method calls, as internally something different happens than described in the documentation. Also some problems regarding identifying different special devices arouse, so testing needed to be postponed, too. The steam controller we wanted to integrate isn't that easily to use as it is registered as a vendor specific device and not a gamepad. The device itself informs the system to be a mouse and a keyboard with the same handle. This would need a custom solution for our Input Handling only to support the Steam Controller, and even then the gyroscope data cannot be read without the Steamworks API, which means a very huge overhead for the whole Engine. We will therefore concentrate on smartphone integration for the Desirable Target.

3. Desirable Target

Overview

The desirable target should include an animated surfer on the surfboard. This also needs a physically correct interaction between the board and the river, also the center of mass of the surfer has a crucial effect on this. We also want to have a highscore list and a main menu to have a better feeling of the application as a game, not just a run once simulation. This means that correct loading and everything regarding that setup needs to be handled. The gameplay element will be a big focus. So balancing the game is a high priority, followed by some advanced special effects of foam and spray induced by the river physics.

4. Design revisions

While most game elements currently only provide basic functionality, we did not change a lot of aspect of our initial game and engine design decisions.
The Input Manager has been reduced to support only the foundation of controlling several device inputs via Raw Input and the game input will be focused on a smartphone application.

5. Challenges

Creating a custom engine requires focus on the key elements on the design. The architecture needs to be easy to understand but at the same time very performant and efficient to realize the graphical challenges and, most importantly, the physically based liquid simulation.
Combining every aspect of the engine to work perfectly together was a big challenge. When we combined the different frameworks and elements of the engine, it worked instantly in a way that the engine can rely on functionality of a certain part of the engine without knowing how it has been realized. This makes the architecture of our Tsunami Game Engine extensible and in that way very powerful.