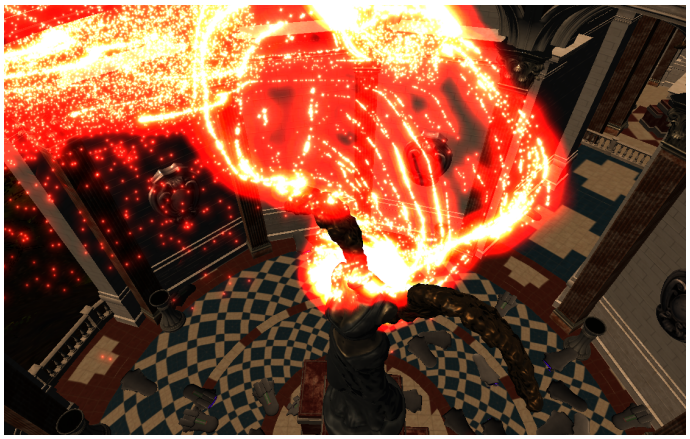
Department of Informatics

Institut für Visualisierung und Datenanalyse,
Chair of Computer Graphics

Praktikum GPU-Computing

Collision Detection using Raytracing Acceleration Structures



Personal Report:
Jonas Heinle April 27, 2022

Contents

List of Figures

List of Tables

1 Proposal

Team project of Jonas Heinle and Sidney Hansen.

Collision Detection using Raytracing acceleration structures:

- For particle systems: Store the scene geometry (triangles) in the acceleration structure and shot rays proportionally to a objects speed in its moving direction, in order to find intersections.
- For grid objects physics simulation:
  Store AABBs in the acceleration structure and shot rays spanning the bounding box, in order to find intersection candidates.

Milestone 1:

- Setting up Stuff (CMake, Vulkan Instance and co)(Jonas)
- Setting up framework (Sidney)
  - Memory Management
  - Abstraction for Vulkan Objects (Pipeline, Descriptor Sets, ...)
  - Importing and Loading Obj Models (Sun Temple, Viking)
  - Texture Loading + Generating Mip Maps
- Implementing Deferred Shading (Sidney)
  - Material System and Shading (Frostbite standard model)
  - Normal Mapping
  - Rendering Particles
- Vector Fields (Jonas)
  - creating 2D Vector fields
  - Implementing 3D Textures for 3D vector fields
- Particle system (Jonas)
  - Creation and uploading particle buffer
  - Implementing integration and simulating pipeline
- Shader hot reload (Both)
- Camera system (Both)

Milestone 2:

- Collision detection and response for particle systems using ray queries (Both)
  - Building acceleration structures (Sidney)
  - Particle collision detection + response (Jonas)
- Working on framework (Sidney)
- Starting with physics simulation (Both)
- ImGui to control Stuff (Jonas)
- (Optional) More types of particle systems (Jonas)

- (Optional) Improving deferred renderer (Sidney)

Milestone 3:

- Physics Simulation for rigid objects (Both)
  - finding hit pairs using established techniques (e.g. sort and sweep / grid)
  - finding hit pairs using ray queries
  - Comparison
- (Optional) Possibility to interact with objects. e.g pick up and throw objects
- (Optional) More types of particle systems (Jonas)

2 First Milestone

2.1 Task 1: Setting up application core (Jonas)

I set up the CMakeLists.txt with all necessary libraries. Subsequently I buit the very Vulkan core (Instance, swapchain and co.). Since this is very straight forward this was done quite fast and we were ready to work on our main tasks.

2.2 Task 2: Set up all compute parts for particle calculations (Jonas)

Since we are interested in finite mass elements getting updated according to the Verlet-Algorithm (see subsection 2.3) I decided to include one **Integration pipeline** and one **Simulation pipeline** (inspired by blog from Sascha Willems [9]). This gives us the flexibility to have one pass which updates the velocity and acceleration according to Verlet and an other pass which we can use for the simulation. Hence we can easily integrate different forms of simulations later on.
We are able to run three dimensional compute shader invocations with arbitrary workgroup sizes (at least as much as our physical device allow us to use) by using spezialition constants.
I have one storage buffer I write my results to and 3 (# swapchain images) storage buffers I subsequently copy the results to. These storage buffers are used in the rendering pipeline after a render-compute semaphore is signaled.

2.3 Task 2: Particle System (Jonas)

For the particle system we are using the **Lagrangian Approach**. This technique implies a discretization of the domain into finite mass elements. Each element (particle) has specific attributes (position and velocity as a starting point). We do now track those particles over time and update their attributes accordingly.
A crucial part of any particle system will be the performed update on the attributes. As a closed form solution is too expensive we are interested in good and cheap numerical solutions. A good starting point is the **Verlet-Algorithm** [7] used in molecular simulations. Giving us the update rules for the position (1) and velocity (2).

$$\vec{x}(t + \Delta t) += x(t) + v(t)\Delta t + 0.5a(t)\Delta t^2 \tag{1}$$

$$\vec{v}(t + \Delta t) = v(t) + 0,5 * (\vec{a}(t) + \vec{a}(t + \Delta t))\Delta t \tag{2}$$

This steps are performed by the integration pipeline (see section 2.2)
This integration step (see 1) and the (for now) simplistic simulation pass are cheap to compute. Together with the deferred rendering approach we are currently able to render millions of moving particles.

```
layout(binding = 0) buffer buf {
        Particle particles[];
};

layout(binding = 2) uniform sampler3D vectorField;

void main() {

    // index into particle buffer for clarity not written out
    int index = ...;

        vec3 current_velocity = texture(vectorField,
                                (particles[index].position.xyz)).xyz;
        vec3 current_color = particles[index].color.xyz;

        // handle particles leaving the field
        if(particles[index].position.x > max_pos.x) {
                current_velocity = vec3(-1.f, 0.0f, 0.f);
        }if(particles[index].position.x > max_pos.x)
        ....
        // handle all cases; this handling is just for now
        // for not having particles leaving the scene

        particles[index].velocity = vec4(current_velocity,0.0);
        particles[index].acceleration += vec4(current_velocity,0.0);
        particles[index].color = vec4(current_color,0.0);

}
```

Figure 1: The simulation step

The simulation pass already uses a 3D texture which contains a vector field. We can now easily access the texture simply with the position. For now we catch particles flying outside with simple if-clauses.
We can pass on the vector velocity from the texture directly to the buffer which will be used in the integration step after it. The acceleration simply is updated by adding the current velocity to it.

2.3.1 Vector Fields

For the particle system we decided to go with vector fields since they are a very common technique [6] for our use case.
Firstly I implemented 2D vector fields and experimented with layering them for the 3-dimensional case. This was rather unsatisfying (E.g. managing the layers, manually interpolating, etc. is exhausting).

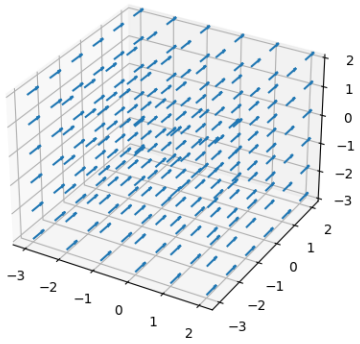


Figure 2: Linear vector field

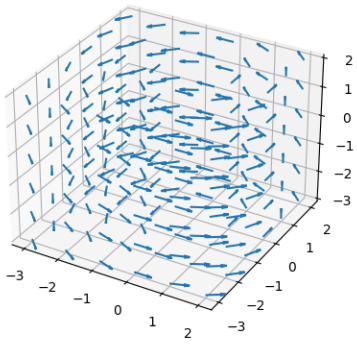


Figure 3: Rotating vector field

Figure 4: Simple 3D-vector fields

After implementing 3D texture support I experimented with with 3D vector fields (see figure 4) and immediately appreciated the benefits (e.g. access the texture directly with 3D position of particle, automatic trilinear interpolation).

2.3.2 Particles

We are now able to upload an arbitrary amount of particles for each dimension separately which are influenced by a given vector field (see figure 5). Also We are able to dynamically add and remove particles in later milestones.

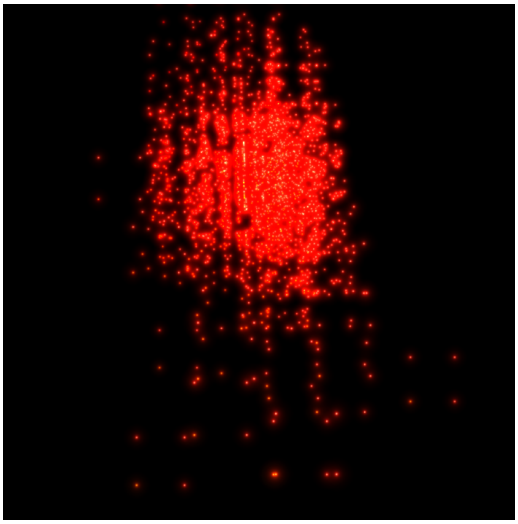


Figure 5: Particles moving in vector field

2.4 Problems (Jonas)

I was able to learn much about Vulkan. It is very interesting to build a compute pass to an rendering application (and e.g. synchronize them, exchanging the particle storage buffer, ...). Nevertheless I encountered quite a few problems.
One tough bug was (see figure 26) the wrong alignment in my my particle struct.

```
struct Particle {
    glm::vec3 position;
    glm::vec3 color;
    glm::vec3 velocity;
    glm::vec3 acceleration;
    glm::vec3 position;
};
```

Figure 6: Not well aligned data

```
struct Particle {
    glm::vec4 position;
    glm::vec4 color;
    glm::vec4 velocity;
    glm::vec4 acceleration;
    glm::vec4 position;
};
```

Figure 7: Well aligned data

Figure 8: Alignment requirements

When writing particles in this manner to an storage buffer one does break alignments requirements. vec3 has to be aligned by 16bytes which they are not. I solved this problem by only using vec4 for every attribute. This problems was so hard to find because no validation layer errors were generated.

Besides not well aligned data the way we implemented shader hot reload gave me access denied errors. It took me a while to realise the space character in my user name (C:/Users/Jonas Heinle/.../shader.comp) was causing problems. Therefore I had to implement an other way for shader hot reload for me.

3 Second Milestone

3.1 GUI

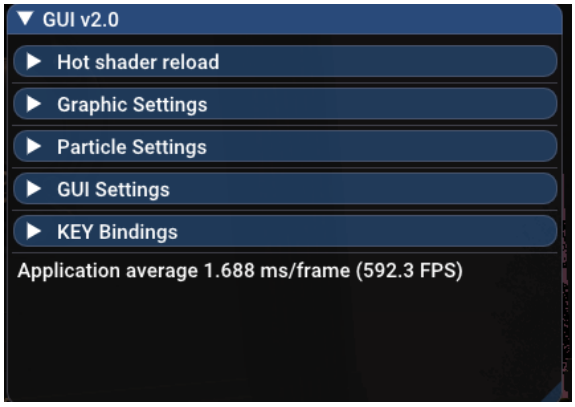


Figure 9: Our GUI

With the GUI we've implemented this milestone (IMGUI [1]) we now can trigger shader hot reload with a button additionally to the possibility of using key shortcuts.
Furthermore I implemented a granular system for controlling the parameters of our particle system. The following figures are describing those parameters and their functionality.

3.1.1 Particles

Figure 10: Main components of the particle system

Vector field
The first sub-category describes our vector field in use. You can change the vector field strategy which determines primarily the movement of the particles. Moreover it is possible to change the size of the vector field thus changing the area of influence.(see figure 11)

Figure 11: Parameters one can play with regarding the vector field

You can also change the so-called strength of the vector field (length of velocity vectors in field).

Number of particles

Figure 12: Control many parameters about our particles

This section allows you to change the amount of particles and their area of influence (i.e spatial expansion).

Workgroup

Figure 13: Parameters one can set for work group sizes

A big part of this milestone was to implement a work group tester allowing us to automatically test different work group sizes and writing results to a csv-File. This data is then used by a python script visualizing the data. (more information see section 3.2)
Worth mentioning is the ability to run the test over the GUI (see figure 13). The user gets feedback over the test progress. After the test the best work group sizes are automatically chosen.

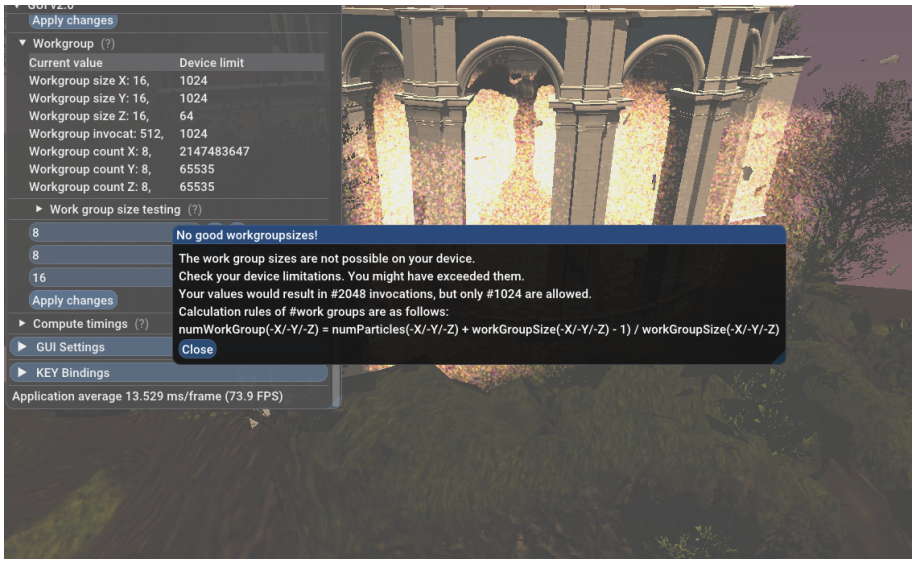


Figure 14: Overlay hinting non-applicable work group sizes

The user gets informed over the individual device limits for the work groups and a pop-up window (see 14) will show up if work group sizes are not possible due to hardware restrictions.
With this the particle system can smoothly fit to different devices.

Timings

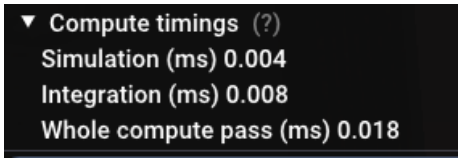


Figure 15: Timer queries for compute shader

We wanted a system to measure both the time of our individual compute stages(i. e. integration and simulation) and the time of the whole compute pass. Hence I implemented time queries for display timing values.

3.2 Work group testing framework

When working with compute shaders it is crucial to take physical device limitations into account (i.e max. amount of invocations) and forbid accordingly non-working working group sizes. I implemented a tester who in a first step examines all possible work group sizes and finally store the results and work group sizes to a csv-file. This file serves as source of a python script exploring and visualizing the data. (using this CSV-Writer [2])

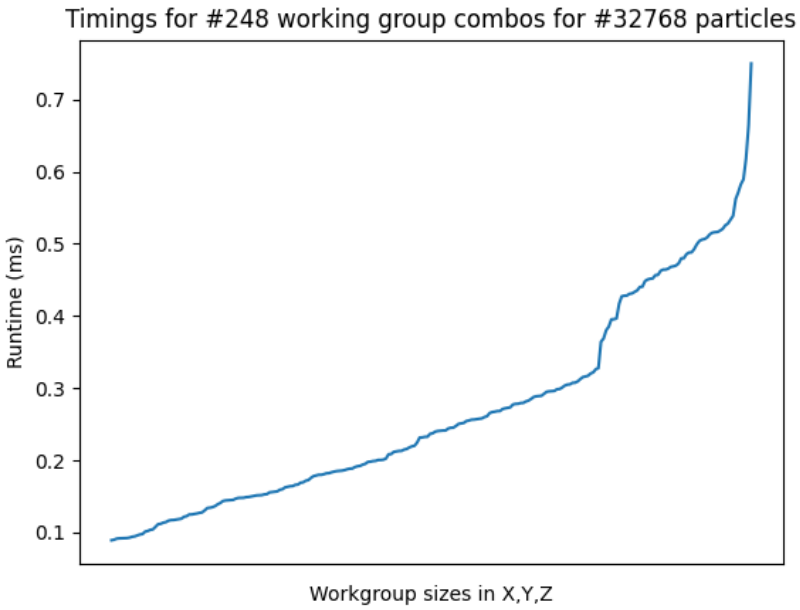


Figure 16: Time over all possible work group sizes

In a first step I was interested in the execution time distribution over all possible work group sizes. The figure 16 shows which execution times are possible (hiding the exact work group sizes).
In a next step I was interested in the best performing work group sizes.

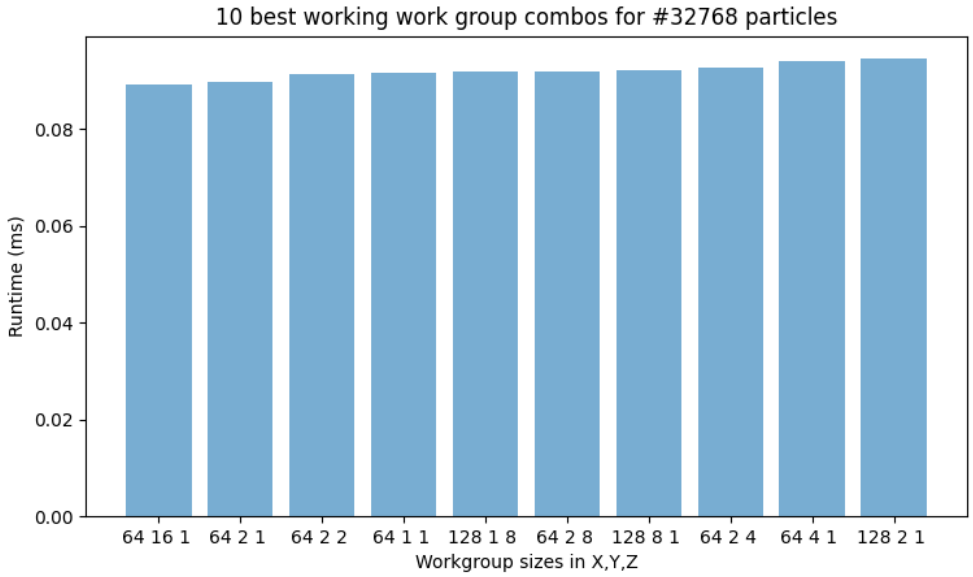


Figure 17: Top 5 work group sizes

The time difference of the best performing work group sizes are neglectable (see figure 17, 0,4% time difference between first and last). But we can see that we benefit from large work group sizes.

Comparing to the worst performing work group sizes (see figure 18) we can see very small work-group sizes.

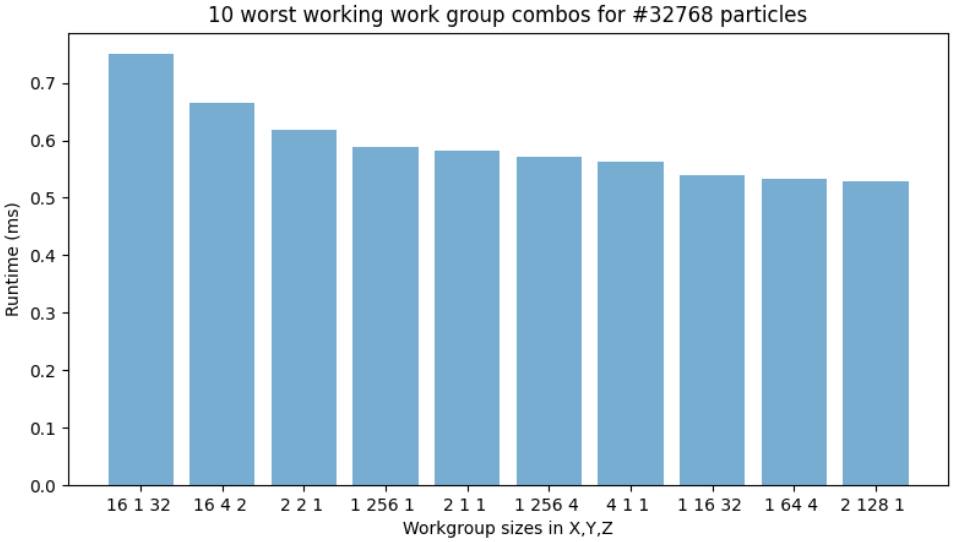


Figure 18: The 5 Worst work group sizes

Interestingly one work group size combination is performing significantly worse than all other. The rest is showing a similar small time difference.
Hence we profit by pretty large work group sizes for now.

3.3 Particle collision detection

With the TLAS created by Sidney I can detect collisions using ray queries against the scene geometry. Letting the particle spawn inside the sun temple those who are hitting scene geometry stop moving (for demonstrating collision detection purposes, see figure 19)

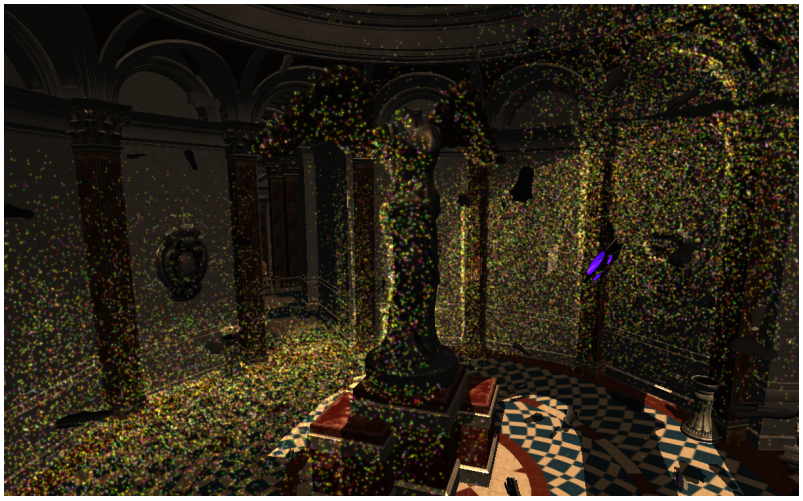


Figure 19: Particles stopping when hitting geometry

The rest of the particle are moving out the archway and moving in the scene (see figure 20).

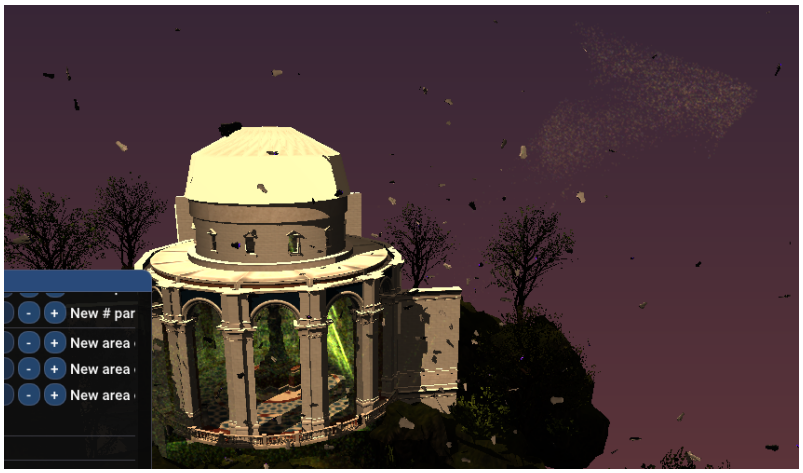


Figure 20: Particles stopping when hitting geometry

With this working collision detection I started to examine different systems for responses. In fig. 21 we can see an exemplary response to a collision event (particle with scene). Here the particles are constantly reflected on the surface normal when trying to leave the scene.

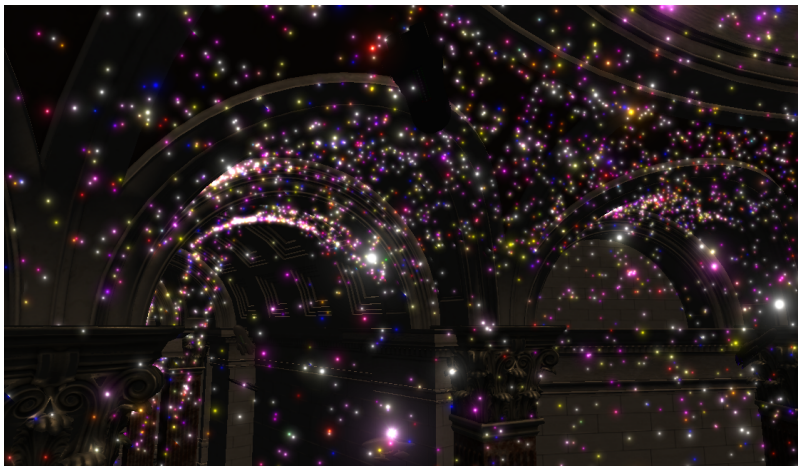


Figure 21: Particles responding to collision detection

4 Third Milestone

4.1 Vector Fields

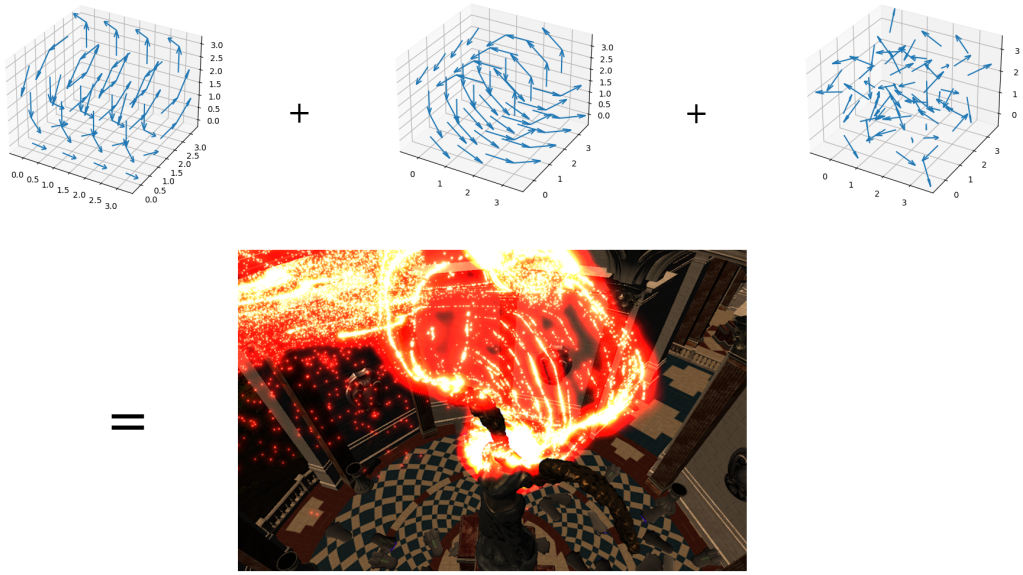


Figure 22: Layering the vector fields

To allow more types of particles I decided to layer vector fields. This gives more degree of freedom in their movement. For example an arbitrary 3d-rotation.(see fig.22)

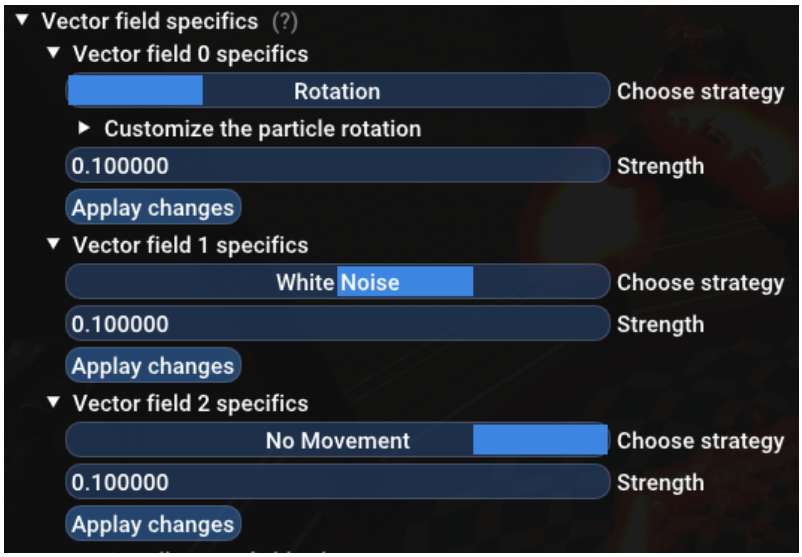


Figure 23: Customize every vector field

All vector fields can be controlled with the UI. (see 23)

4.2 Fixing work group size tester

I was fixing the work group size tester and updated the figures 16 17 18 from the previous milestone accordingly. We discussed the work group sizes the last milestone and they are indeed not possible. I was wrongly using maxComputeWorkGroupCount instead of the maxComputeWorkGroupSize ...

4.3 Collision Detection

For implement continuous collision detection I use ray queries. (source: [10]). Having the acceleration structure already built I can simply shoot a ray in the direction of its velocity against the scene. The particle response strategy (see fig. 24) will decide how the particle will move further. I used the glue strategy for debugging purposes mostly. For this strategy the new particle position will just be set to the intersection position and its velocity to zero.
The more interesting strategy will be the "bounce". Corresponding to this strategy the new particle position will be set to the collision point with its velocity and acceleration becoming reflected.

Figure 24: Layering the vector fields

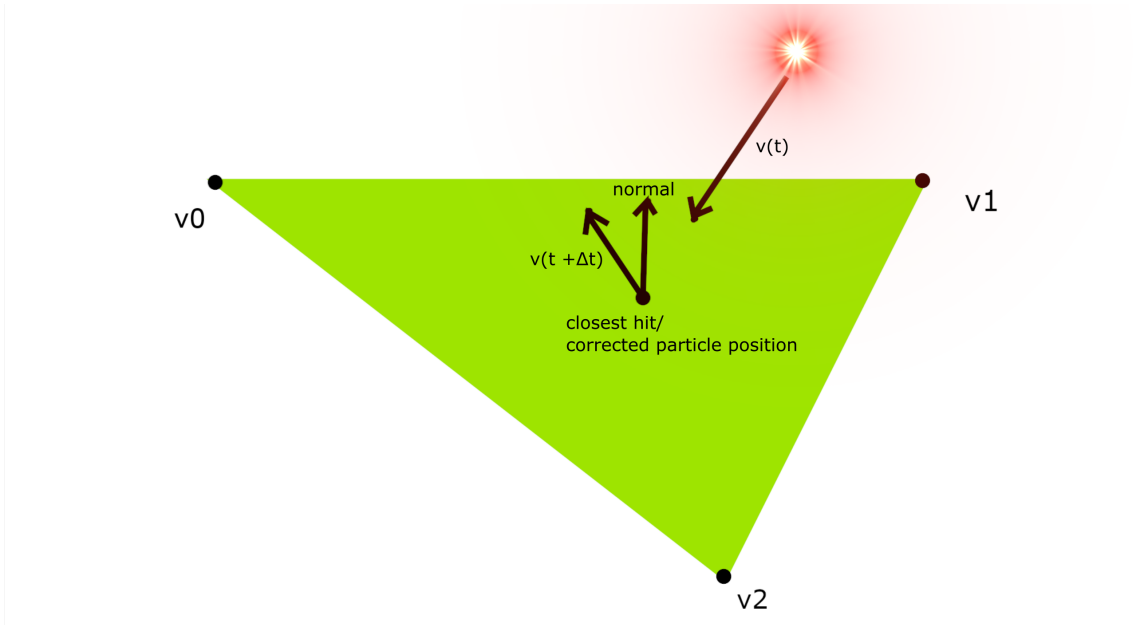


Figure 25: Layering the vector fields

The figure 25 shows the situation how a particle gets reflected. What we get for free from a ray query (if it finds a hit) are the barycentrics and the involved vertices. With this information we can construct the normal at the closest hit and reflect the particle velocity and acceleration against it.

```
// retrieve instance id from hit
int instID = rayQueryGetIntersectionInstanceCustomIndexEXT(rayQuery,true);
// retrive adresss from the previously uploaded buffers
uint64_t verticesAddr = inst[0].vertices;
uint64_t indicesAddr = inst[0].indices;
// with the addresses we can get the vertices and indices
Vertices vertices = Vertices(verticesAddr);
Indices        indices        = Indices(indicesAddr);
ivec3 ind = indices.i[instID];
Vertex v0 = vertices.v[ind.x];
Vertex v1 = vertices.v[ind.y];
Vertex v2 = vertices.v[ind.z];
// get barycentrics from the rayQuery
vec2 baryzentric = rayQueryGetIntersectionBarycentricsEXT(rayQuery, true);
const vec3 barycentrics        = vec3( 1.0 - baryzentric.x - baryzentric.y,
                                 baryzentric.x,
                                 baryzentric.y);
// reconstruct normal with the barycentrics
vec3 modelNormal =  v0.normal * barycentrics.x +
                    v1.normal * barycentrics.y +
                    v2.normal * barycentrics.z;
// instances buffer also previously uploaded
vec3 N        = (inst[instID].modelMat*vec4(modelNormal, 0.0)).xyz;
```

Figure 26: What happens at a hit

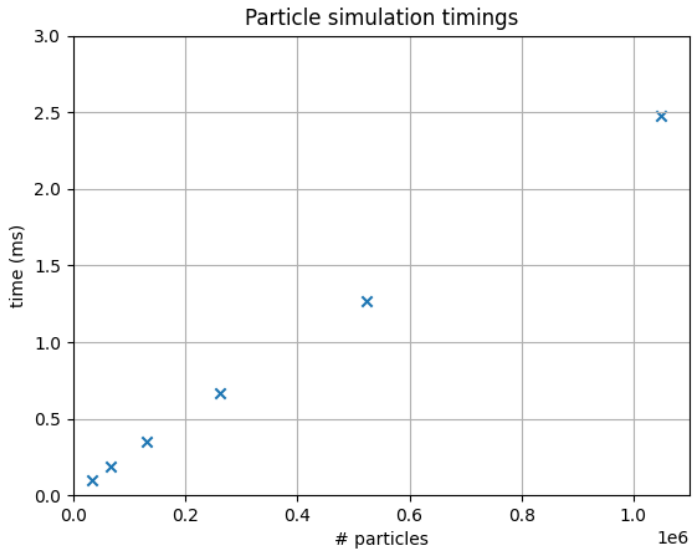


Figure 27: Timings of the particle compute stage

All calculations regarding the particles (see fig. 27) are pretty fast. Even for over a million particles the timings stay under 2,5ms/frame.

4.4 Rigid Body Simulation

In this milestone we compared our method with a established technique. We decided to use a grid based approach for this test [5].
For this collision detection approach [5] we needed to generate a particle based representation from an object. You can think of a 3d-uniform grid defining cells. When a particle overlaps with the geometry we store the position.

4.4.1 Generating object particles

I decided to generate the particles in two main steps. The first step **generates the silhouette particles**. Subsequent to the generation there is a **sort and fill** step. Both steps are inspired by Order Independent Transparency techniques [4] [8].
I decided against the depth peeling approach [3] the GPU Gems article uses for generating the particles [5]. Since depth peeling needs multiple scene passes I hope to improve speed in comparison to them.

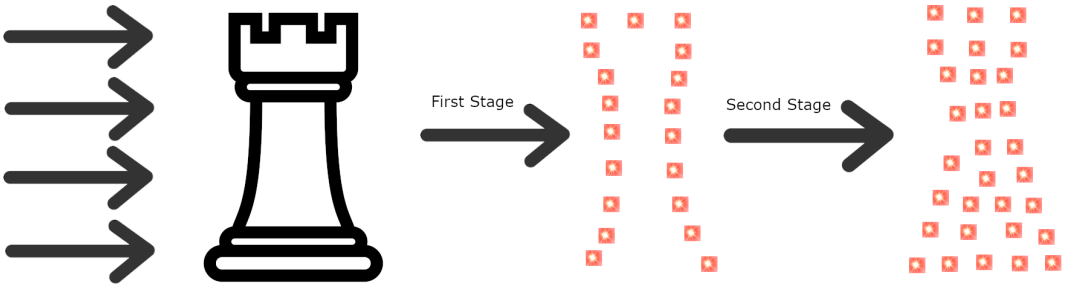


Figure 28: Object particles without insertion

First stage: Generate silhouette particles
For generating silhouette particles I decided to use per-pixel linked lists (commonly used in f.e. order independent transparency [8]). If the cull mode is disabled this technique guarantees that no fragment will be discarded as early fragment tests are disabled. A screen wide image stores the head pointer of a linked list. For appending new items to the list I am using atomic counters as uniform variables. The corresponding object will be rasterized. It is viewed by the left side and we use an orthogonal projection (as described in [5]). The figure 29 is showing the generated silhouette particles. One can see the occurring problems quite clear. The orthogonal projection leaves big holes in the generated particles. We need to fill it up.

Second stage: Sort and fill
In the first step we only generated the silhouette particles. But we need to fill the 3-dimensional grid. Otherwise we will run in problems (see f.e. fig. 29). We want to fill the space between to neighboring generated particles with a proper number of new particles. Hence we need to sort the generated particles with respect to their depth. Then we can generate new particles uniformly between them. The figure 30 is showing the final result of the generated object particles. The holes are filled with particles.

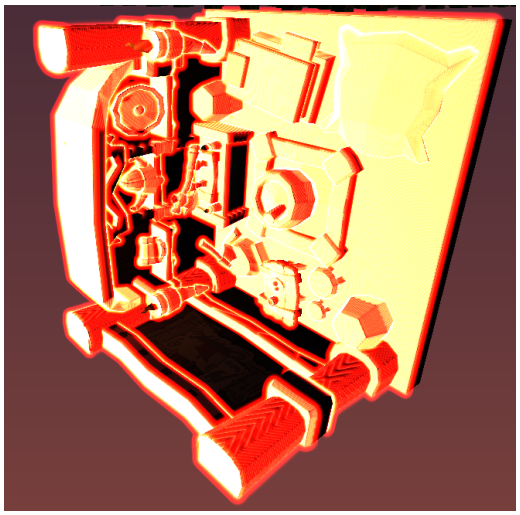


Figure 29: Object particles without insertion

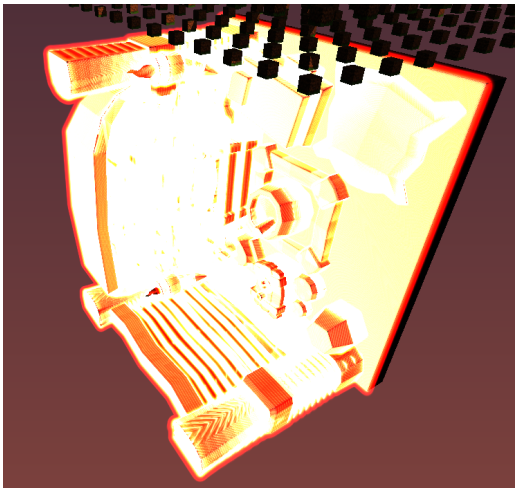


Figure 30: Object particles with insertion

Figure 31: Alignment requirements

References

[1] GUI contributors. *GUI*. URL: https://github.com/ocornut/imgui (visited on May 2, 2021).

[2] al-eax. *CSV file writer*. URL: https://github.com/al-eax/CSVWriter (visited on Dec. 19, 2021).

[3] Cass Everitt. *Depth peeling*. URL: https://www.gamedevs.org/uploads/interactive-order-independent-transparency.pdf.

[4] Nvidia. *Order-Independent Transparency*. URL: https://on-demand.gputechconf.com/gtc/2014/presentations/S4385-order-independent-transparency-opengl.pdf.

[5] Nvidia. *Real-Time Rigid Body Simulation on GPUs*. URL: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-29-real-time-rigid-body-simulation-gpus.

[6] Unreal. *Vector Fields Unreal Engine*. URL: https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/ParticleSystems/VectorFields/.

[7] Wikipedia. *Verlet-Algorithmus*. URL: https://de.wikipedia.org/wiki/Verlet-Algorithmus.

[8] Sascha Willelms. *Order-Independent Transparency*. URL: https://github.com/SaschaWillems/Vulkan/tree/master/examples/oit.

[9] Sascha Willems. *Blog of Sascha Willems*. URL: https://github.com/SaschaWillems/Vulkan/tree/master/examples.

[10] Sascha Willems. *Ray queries in GLSL*. URL: https://github.com/KhronosGroup/GLSL/blob/master/extensions/ext/GLSL_EXT_ray_query.txt.