TU WIEN Informatics

Particle System in WebGPU

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Benedikt Peter
Registration Number 00171283

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Dr.techn. BSc Markus Schütz

Vienna, 3rd February, 2023

Benedikt Peter	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Benedikt Peter

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Februar 2023

Benedikt Peter

Abstract

"In this thesis, we investigate the performance of particle system simulation and rendering in WebGPU. A prototype that can produce various particle effects in the browser was implemented using the WebGPU API. The particle's lifetimes and movements are simulated on the GPU using a compute shader, and rendered as textured quads. By using a compute shader, it is possible to simulate and render ten million particles at around 63 frames per second (on a GTX 1060). For rendering the particles, both *instancing* and *vertex pulling* were implemented, and a comparison between the two modes shows that on recent high end GPUs, using vertex pulling leads to significantly better performance than using instancing."

Contents

CHAPTER 1

Introduction and related work

For this bachelor thesis, a prototype for a particle system in the new WebGPU API was developed. The goal was to implement a system in which various different parameters can be changed in order to test how they affect performance. Using timestamp queries, benchmarks were performed to measure the frame times for different particle amounts and sizes, as well as to determine whether using instancing or vertex pulling is more efficient for the purpose of rendering particles.

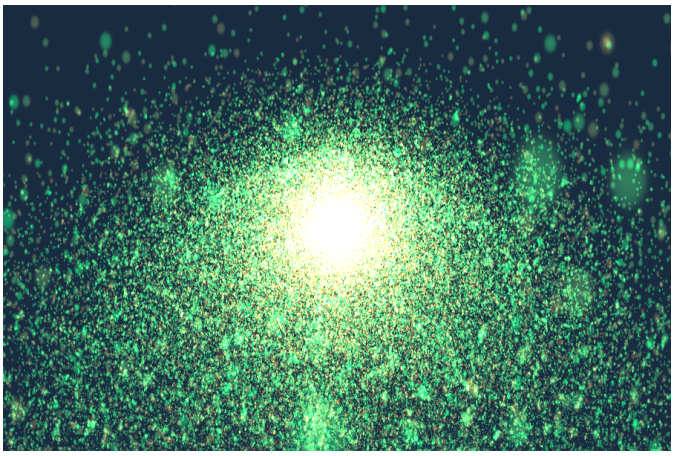


Figure 1.1: A particle effect created using the prototype.

1.1 Particle Systems

A particle system is a method used in computer graphics to simulate various effects, such as explosions, sparks or smoke. They were first used in the 1982 film "*Trek H: The Wrath of Khan*", and the model was described in a 1983 paper by Reeves[17].Unlike methods used

for rendering solid geometry, where geometry is defined by polygons, objects created with particle systems are defined as *clouds of primitive elements* (particles). These particles spawn and expire, and during their lifetime they can change in position or appearance. In every frame new particles are initialized, expired particles are removed, the particles' attributes are updated and the scene is rendered. The rendering is conventionally done using rasterization, but recently more efficient methods that make use of modern GPU's path traversal capabilities have also been developed[16]. Due to the increasing computational power of GPUs in the 2000s, in addition to the rendering, the simulating of the particles is today also often done on the GPU[11]. Compute Shaders allow asynchronous execution of various tasks, which allows the simulation and rendering of large amounts of particles at great performance[12]. The prototype described in this thesis uses the WebGPU API to simulate and render particles, with the simulation being done on the GPU using a *compute shader*.

1.2 WebGPU

Since 2011 *WebGL*, a JavaScript API based on OpenGL ES, has made rendering 3D graphics without the use of plugins possible in compatible browsers[2]. The *WebGPU* API, which is currently being developed by the *GPU for the Web Community Group*, aims to be a successor to WebGL, but unlike WebGL it is not a port of an existing API[2]. WebGPU is being developed with a bigger focus on enabling the usage of the GPU for more general computations, rather than just for rendering[1]. WebGPU provides access to the GPU via GPUAdapter, GPUDevice and GPUQueue objects, with commands being defined via a GPUCommandBuffer that gets submitted to the GPUQueue. The programs executed on the GPU are described by pipelines, which are a combination of fixed-function stages and shaders[5]. The shaders are written in the shading language WGSL, and support three types of entry point functions, *vertex*, *fragment* and *compute*[7]. For this prototype, the vertex and fragment shaders are used for rendering the particles while a compute shader is used for simulating the particles' life cycles and movements.

CHAPTER 2

Method

The program is made of a WebGPU canvas element on which the particles are rendered, as well as a GUI that allows the user to control various parameters and settings. In order to allow the creation of different types of effects, there are three modes that the user can choose from: *default*, *snow* and *tree*. These modes affect how new particles are spawned. In the *default* mode they spawn at a fixed origin point and shoot in random directions, while in the *snow* and *tree* modes they spawn on a layer or within a sphere around the origin respectively, and then fall according to the gravity.

The process of rendering particles consists of two stages that are executed once per frame: simulating the particles and rendering them on the screen. In the simulation stage, the particles' velocities are calculated and applied to the position. The spawning and expiration of particles is also handled in this stage. Applying these simple calculations to millions of particles on the CPU side is inefficient, which is why a *compute shader* is used instead to perform these calculations on the GPU. This way one can take advantage of the GPU's large number of parallel processors to significantly accelerate this process.

In terms of their actual geometry, every particle that is drawn on the screen is a quad, consisting of two triangles. In this prototype two different approaches to rendering the particles, *instancing* and *vertex pulling*, were both implemented.

The *particles* sample from Austin Eng's WebGPU Samples[13] was used as a reference for the implementation of the prototype.

2.1 Prototype Structure

2.1.1 Setting up the WebGPU device and context

When the program starts, a *Renderer* object is created and the *Renderer.initRenderer* function is called, which sets up the WebGPU context and creates all relevant buffers

and pipelines. At first the *GPUAdapter* is requested, then, when requesting the device, the following requirements are added to the descriptor:

```
let deviceDescriptor : GPUDeviceDescriptor = {
        requiredLimits: {
                maxStorageBufferBindingSize : 512 * 1024 * 1024, // 512mb
            },
        requiredFeatures: ["timestamp-query"]
        };
 (...)
 this.device = await adapter.requestDevice(deviceDescriptor);
```

The *maxStorageBufferBindingSize* is increased to 512 megabytes since by default the maximum storage buffer size is only 128 megabytes. This greatly increases the number of particles that can be rendered. The feature *timestamp-query* is required for precisely measuring the execution time of WebGPU passes, which is needed for the benchmarks. If the browser does not support or allow timestamp-queries, a *GPUDeviceDescriptor* without this property will be used instead. Then a *GPUContext* is created for the canvas defined in the html file, and in its configuration the *device* is set to the *GPUDevice* requested earlier, the *format* is set to *"bgra8unorm"* and the *alphaMode* is set to *"opaque"*.

After this, the pipelines and buffers for the compute, vertex and fragment shaders are created. This will be described in Sections 2.2.1 and 2.3.1.

2.2 Simulating Particles

A particle is defined by four properties: its position, its velocity, it's right vector and its remaining lifetime. In every frame, the gravity is applied to the velocity, the velocity and the wind are applied to the position, the right vector is rotated, and the remaining lifetime is reduced. When a particle's lifetime reaches zero, it despawns and a new one can spawn in its place. These steps need to be applied to every particle once per frame. Storing millions of particles in an array and using a loop to iterate over all of them and simulate them on the CPU is not feasible, which is why a compute shader is used. All the information about the particles is stored in a GPU buffer and a compute shader modifies this data on the GPU.

2.2.1 Compute Pipeline

In the compute shader, the particle buffer is interpreted as a struct of the type *Particles*, which is defined as follows:

```
struct Particle {
    position: vec3<f32>,
    lifetime: f32,
    velocity: vec3<f32> ,
    rightRotation: vec3<f32>
}

struct Particles {
  particles : array<Particle>
};
```

We can see that every particles takes 32 bytes of data (one float of 4 bytes and three three-dimensional vectors of 12 bytes each). Due of WebGPU's memory layout requirements[9] there needs to be an additional 4 bytes of padding after the velocity and after the rotated right vector in the buffer, so one particle takes up 48 bytes.

The particle buffer is created like this:

```
this._particleBuffer = this._device.createBuffer({
            size: this._numParticles * Particles.INSTANCE_SIZE,
            usage: GPUBufferUsage.VERTEX | GPUBufferUsage.STORAGE
});
```

The constant *INSTANCE_SIZE* is 48, and this number is multiplied with the number of particles to get the required buffer size. The buffer usage flag *STORAGE* is needed so that the compute shader can retrieve and store data from the buffer, and the flag *VERTEX* is required so the same buffer can also be used as a vertex buffer (as shown in Section 2.3.1).

The creation of the pipeline for the compute pass is analogous to creating a render pipeline:

```
const computePipelineDescr : GPUComputePipelineDescriptor = {
            layout: 'auto',
            compute: {
                module: this._device.createShaderModule({
                    code: simulationComputeShader
                }),
                entryPoint: 'simulate'
            }
        }
 this._simulationPipeline =
        this._device.createComputePipeline(computePipelineDescr);
```

The compute shader also requires a uniform buffer that contains various parameters needed for the simulation, which will be explained in Section 2.2.3.

Finally a bind group that binds the particle buffer and the uniform buffer is created.

2.2.2 Update function

The *update* function is called every frame before the render pass. At the beginning of the function, the uniform buffer is updated, and deltatime is passed to the function as an argument, the random seed is generated by calling *Math.random()* four times, and the remaining parameters are either constants or are retrieved from the GUI. When the number of particles is changed, the particle buffer size has to be changed. Resizing GPU buffers is not possible, therefore a new buffer has to be created. Once the buffers are updated, the command encoder for the compute pass is created and submitted. A *GPUCommandEncoder* object is created using *GPUDevice.createCommandEncoder*, and a compute pass is started with *GPUCommandEncoder.beginComputePass*. After the pipeline and the bind group have been set, the workgroups are dispatched to perform the compute shader program on them:

```
passEncoder.dispatchWorkgroups(Math.ceil(this._numParticles / 256))
```

Compute shaders allow the GPU to concurrently execute a program. The set of invocations that can be executed simultaneously is called a *workgroup*.[8] We use the maximum possible size for a workgroup, which is 256 (this size is set in the compute shader). Every workgroup simulates 256 particles concurrently, which means the number of workgroups we need to dispatch to process all particles is equal to the number of particles divided by 256. After encoding the *dispatchWorkgroups* command the command encoder can be submitted to the device queue and the compute shader will be executed over all particles.

2.2.3 Compute Shader

A compute entry point in a WGSL shader is defined with the *@compute* attribute. It also requries the *@workgroup_size* attribute, which defines the work group dimensions. Since a one-dimensional work group with a size of 256 is used in this program, only the number 256 is passed as a parameter. The id of the invocation can be retrieved by using the built-in value *global_invocation_id* as a function argument.

```
@compute @workgroup_size(256)
fn simulate(@builtin(global_invocation_id) GlobalInvocationID : vec3<u32>) {
(...)
}
```

The x-coordinate of the *GlobalInvocationID* is used as an index to access the individual *Particle* objects in the *particles* buffer. The particle data is retrieved, modified, and at the end of the function stored in the buffer again.

```
let idx = GlobalInvocationID.x;
// load particle from buffer
var particle = data.particles[idx];
(...)
// write updated particle data into buffer
 data.particles[idx] = particle;
```

To simulate a particle's movement in a frame, first the gravity has to be applied to the velocity, then the velocity and the wind are applied to the particle's position. The gravity, velocity and wind have to be multiplied with delta time before being applied, in order to make the simulation frame rate independent. Also, the particle quad's right vector is rotated. This rotation is done using a quaternion, with the rotation axis being a normal to the velocity and the wind vectors and the angle being dependent on delta time. Since WGSL has no built in functions for creating quaternions or using them for rotation, the required function need to be included in the shader code. The function for quaternion creation, multiplication and rotation were taken from Geeks3D[15] and adapted for WGSL syntax.

```
// apply gravity
particle.velocity = particle.velocity + (params.gravity * params.deltaTime);

// apply wind
 particle.position += params.wind.xyz * params.wind.w * params.deltaTime;
```

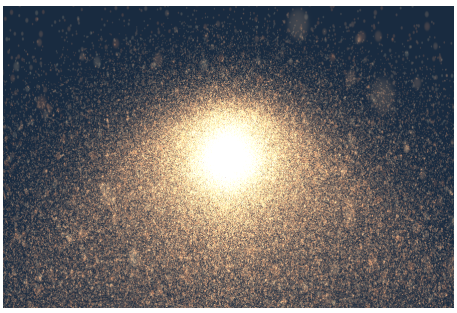
```
// update particle data
particle.position = particle.position + (particle.velocity * params.deltaTime);
particle.lifetime = particle.lifetime - params.deltaTime;
```

The compute shader also needs to handle the spawning of particles. If a particle's lifetime is equal to or below zero (which is always the case at the beginning since the buffer is initialized with all zeroes), the particle's position and velocity are reset according to the mode chosen in the GUI:

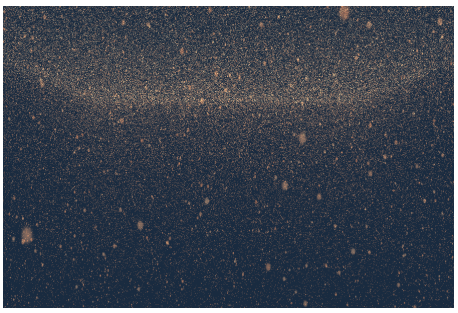
1. In the **Default** mode, the particle's position is set to the origin position specified in the uniform buffer. The velocity is pseudorandomly generated and then its length is randomly scaled with the range being specified by the *initialVelocity* uniform.

2. In the **Snow** mode, the position is randomly picked within a range on a horizontal plane at the height of the *origin* uniform. The velocity is also randomly generated, but is scaled with a very small length, to make the movement seem slightly randomized.

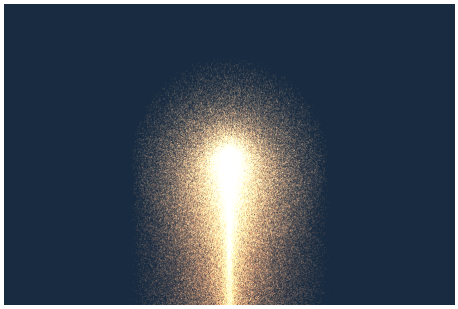
3. In the **Tree** mode, the particles are spawned within a sphere around the *origin*. The velocity initialization is the same as in the *Snow* mode.



(a) The *default* mode.



(b) The *snow* preset.



(c) The *tree* preset.

Figure 2.1: The prototype's three spawn modes.

The buffer also contains a minimum and maximum lifetime, to pick the particle's initial lifetime a random number between 0 and 1 is generated to interpolate between those to numbers.

Since all particles start at a lifetime of zero, all of them would normally spawn in the first frame, which causes an explosion-like effect when the site is loaded. To prevent this, the number of spawns per second is capped. To allow this, the number of spawns needs to be counted, which requires an atomic unsigned int. The use of *AtomicLoad* in all invocations of the shader program however causes a notable decrease in performance. To circumvent this, buffer aliasing is used, meaning there exist both an atomic and a non atomic binding, with the same buffer being bound to both.

```
@binding(2) @group(0) var<storage, read_write> spawnCounter : atomic<u32>;
@binding(3) @group(0) var<storage, read_write> spawnCounterNonAtomic : u32;
```

Simply reading the value from the non-atomic binding greatly mitigates the performance cost of using an atomic buffer. Both the particle cap and the buffer aliasing can be toggled on or off in the GUI, which allows the user to compare the performances.

Since WGSL contains no built in pseudorandom number generator function, one needs to be included in the shader. The random function used in this program was taken from the Austin Eng sample[13].

```
var<private> rand_seed : vec2<f32>;

fn rand() -> f32 {
  rand_seed.x = fract(cos(dot(rand_seed, vec2<f32>(23.14077926, 232.61690225)))
        * 136.8168);
  rand_seed.y = fract(cos(dot(rand_seed, vec2<f32>(54.47856553, 345.84153136)))
        * 534.7645);
  return rand_seed.y;
}
```

At the beginning of the *simulation* function, the seed is calculated, using a combination of the seed uniform (which does not change within a frame) and the particle's index to create a unique seed for every particle.

2.3 Rendering Particles

After the particles' positions have been updated in the compute shader, in the rendering stage quads are rendered in those positions. In this project, two approaches, *instancing* and *vertex pulling*, can be used to render the particles. Since the geometry of all particles is identical (a quad made of two triangles), instancing can be used to draw all particles, with the particle data being retrieved from a vertex buffer. In the *vertex pulling* approach, the particle data is retrieved from the particle buffer using the vertex index.

2.3.1 Rendering Pipelines

Because of how pipelines work in WebGPU, it is possible to create two different render pipelines, one using *instancing* and one using *vertex pulling*, that both use the same

vertex shader. This is done by using different entry point functions of the shader and by defining the buffers differently. The vertex section of the instancing pipeline has the following layout:

```
this.particleRenderPipelineInstancing = this.device.createRenderPipeline({
            layout: "auto",
            vertex: {
                module: this.device.createShaderModule({
                    code: particleQuadVertexShader
                }),
                entryPoint: "main_instancing",
                buffers: [{
                        // instanced particles buffer
                        arrayStride: Particles.INSTANCE_SIZE,
                        stepMode: 'instance',
                        attributes: [ {
                                // position
                                shaderLocation: 0,
                                offset: 0,
                                format: 'float32x3',
                            }, {
                                // lifetime
                                shaderLocation: 1,
                                offset: 3*4,
                                format: 'float32'
                            }, {
                                // rotated right vector
                                shaderLocation: 2,
                                offset: 8*4,
                                format: 'float32x3'
                            }
                        ],
                    }
                ]
            },
                (...)
});
```

Since the particles buffer can also be used as a vertex buffer (see Section 2.2.1), no new vertex buffer has to be created. The array stride is set to the size of a single particle's data in the block, and the position (needed in the vertex shader) and the lifetime (needed in the fragment shader) can be read in the vertex shader. The buffer's *stepMode* property is set to *'instance'* to specify that every array entry (particle) represents an instance.

The vertex pulling pipeline on the other hand is defined as follows:

```
this.particleRenderPipelineVertexPulling = this.device.createRenderPipeline({
            layout: "auto",
            vertex: {
                module: this.device.createShaderModule({
                    code: particleQuadVertexShader
                }),
                entryPoint: "main_vertex_pulling"
            },
           (...)
});
```

A different entry point is used, and since the vertex pulling method does not use a vertex buffer no buffers have to be described.

The fragment sections of the pipeline differ depending on of additive blending should be enabled or not.

```
(...)
fragment: {
                (...)
                targets: [{
                    format: this.format,
                    blend: useAdditiveBlending ? additiveBlending : noBlending
                    }
                }]
            },
(...)
```

The program picks one of these two blend settings::

```
let additiveBlending = {
            color: {
                srcFactor: 'src-alpha',
                dstFactor: 'one',
                operation: 'add',
            },
            alpha: {
                srcFactor: 'zero',
                dstFactor: 'one',
                operation: 'add',
            },
        };

let noBlending = {
            color: {
                srcFactor: 'one',
                dstFactor: 'zero',
                operation: 'add',
            },
            alpha: {
                srcFactor: 'one',
                dstFactor: 'one',
                operation: 'add',
            },
        };
```

Whenever additive blending gets enabled or disabled in the GUI, the pipeline is recreated with the respective blend settings.

2.3.2 Draw call

To draw the particles, a command encoder is created, a render pass started, and the uniform bind groups set. There are also differences between the two modes. When using the instancing pipeline, since every particle has six vertices, the vertex count is set to six and the instance count is equal to the number of particles:

```
renderPass.draw(6, this.particleSystem?.numParticles, 0, 0);
```

For the vertex pulling pipeline, only one instance is drawn, and since every vertex of every particle must be drawn in that single instance, the vertex count is set to the number of

particles multiplied by 6. Also, before the draw call, a bind group containing the particle buffer also used in the compute shader must be set:

```
renderPass.setBindGroup(1, this.particleBufferBindGroup as GPUBindGroup);
renderPass.draw(<number>this.particleSystem?.numParticles * 6, 1, 0, 0);
```

2.3.3 Vertex Shader

Since the vertex shader needs to directly access the *particles* buffer if vertex pulling is used, a binding for it needs to be included. The *Particles* binding is defined identically to the compute shader. When using instancing, this bind group does not have to be set.

```
@binding(0) @group(1) var<storage, read> particleBuffer : Particles;
```

The vertex shader has two different entry points, which retrieve the vertex data and local index and call the *mainVert* function to perform the vertex shader transformation.

```
@vertex
fn main_instancing(vertexInput: VertexInput,
        @builtin(vertex_index) VertexIndex: u32) -> VertexOutput {
    return mainVert(vertexInput.position, vertexInput.lifetime, VertexIndex,
    vertexInput.rightRotated);
}


@vertex
fn main_vertex_pulling(@builtin(vertex_index) vertexIndex: u32) -> VertexOutput {
    let particleIdx = u32(vertexIndex/6);
    let quadIdx = vertexIndex % 6;
    let particle = particleBuffer.particles[particleIdx];

    return mainVert(particle.position, particle.lifetime, quadIdx,
    particle.rightRotation);
}

fn mainVert(particlePos: vec3<f32>, particleLifetime: f32, quadVertIdx: u32,
rightRotated: vec3<f32>) -> VertexOutput {
            (...)
}
```

In the *mainVert* function, the positions of the particle quad's corners in camera space is calculated.

First, the relative position of the corner vertices from the quad's center must be retrieved. As shown in Figure 2.2, the quad is made of two triangles, so the six vertices making up the quad's two triangles are specified in this array:

```
var quadPos = array<vec2<f32>, 6>(
    vec2<f32>(-halfwidth, halfheight),   //tl
    vec2<f32>(-halfwidth, -halfheight),  //bl
    vec2<f32>(halfwidth, -halfheight),   //br

    vec2<f32>(halfwidth, -halfheight),   //br
    vec2<f32>(halfwidth, halfheight),    //tr
    vec2<f32>(-halfwidth, halfheight));    //tl
```

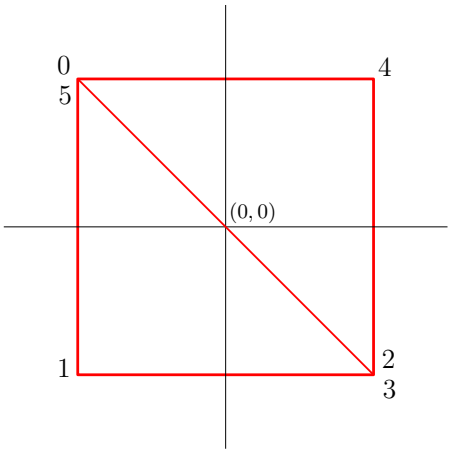


Figure 2.2: The particles are rendered as two dimensional textured quads that always face the camera.

The positions in the *quadPos* array, are accessed using the *mainVert* function argument *quadVertIdx*, which is retrieved by the built-in value *vertex_index* (modulo 6 when using vertex pulling), as the index.

If rotation is not enabled then the particle quads should face the camera. Having two-dimensional sprites always face the camera is called *billboarding*. To achieve this, the camera's up and right vectors in world space are used to calculate the quad's corner positions so that the quad is facing towards the camera. The camera's up and right vectors are retrieved from the view-projection-matrix (which is passed to the shader as a uniform). The right and up vectors can be found in the first and second column.

```
let quadRight = vec3<f32>(camera.viewProjectionMatrix[0].x,
        camera.viewProjectionMatrix[1].x, camera.viewProjectionMatrix[2].x);
let quadUp = vec3<f32>(camera.viewProjectionMatrix[0].y,
        camera.viewProjectionMatrix[1].y, camera.viewProjectionMatrix[2].y);
```

If rotation is enabled, the rotated vector passed by the compute shader is used as the right vector, and the quad's up vector is calculated using the cross product of the rotated right vector and the camera's right vector. The function contains two arrays of length six, one of which contains the relative coordinates of all the quad's corners, and another one which contains their texture coordinates. The quad position's x and y-coordinates are multiplied with the quad's right and up vector respectively and then added onto the particle's position. Then the new position is multiplied with the view-projection-matrix, and is then used as the vertex shader's position output, while the lifetime and the texture coordinate are passed to the fragment shader.

```
if(uniforms.rotationEnabled == 1 && abs(dot(rightRotated, quadRight)) <= 1.0) {
                quadUp = normalize(cross(rightRotated, quadRight));
                quadRight = rightRotated;
            }
```

There is also a different mode, which can be selected in the GUI. In the *pixel sizes* mode, the particle dimensions are given in pixels, and the size of the quads are independent of their positions. To achieve this, the quad offsets are added to the position's x and y coordinates after the position has been transformed. Before adding the offsets, the positions x, y and z coordinates must be divided by positions.w, and positions.w set to 1, otherwise the particle's sizes will be changed afterwards when the division by w is done automatically.

2.3.4 Fragment Shader

The fragment shader is very straightforward. First, the texture color is sampled using the UV coordinates passed from the vertex shader. The circle textures used in the prototype were taken from *Kenney's Particle Pack*[3] and the *leaf1* texture was taken from a license-free set on *freepik*[10]. There are two colors for the particles that the user can set, one for when the particle's lifetime is at the maximum and one for when it is zero. Two color values are created by multiplying the texture color with both of these colors. Then the program linearly interpolates between the two colors depending on it's current lifetime, and returns that color.

```
@fragment
fn main(@location(0) uv: vec2<f32>, @location(1) lifetime: f32)
        -> @location(0) vec4<f32> {
    var textureColor : vec4<f32> = textureSample(textureData, textureSampler, uv);

    if(textureColor.a < 0.01) {
        discard;
    }

    var colorWeight = lifetime / particleUniforms.maxLifetime;
    var maxLifetimeColor = textureColor * particleUniforms.color;
    var minLifetimeColor = textureColor * particleUniforms.color2;

    // interpolate between the two colors
    var fragColor = maxLifetimeColor * colorWeight +
        minLifetimeColor * (1.0-colorWeight);
    fragColor.a = textureColor.a * particleUniforms.alphaFactor;
    return fragColor;
}
```

The alpha component is multiplied with a factor that can be changed in the GUI to change the particle's brightness. In case additive blending is deactivated, transparency can still be achieved by discarding pixels where the texture's alpha value is (close to) zero. This is useful for textures like the leaf texture shown in Figure 2.3a, for which the transparent background is discarded. For textures with alpha-values between 0 and 1, like the circle texture shown in Figure 2.3b, additive blending leads to a more natural look.

(a) The leaf1 texture, one of the selectable texture for the particles.

(b) The circle_05 texture, one of the selectable texture for the particles.

Figure 2.3: Two of the selectable textures.

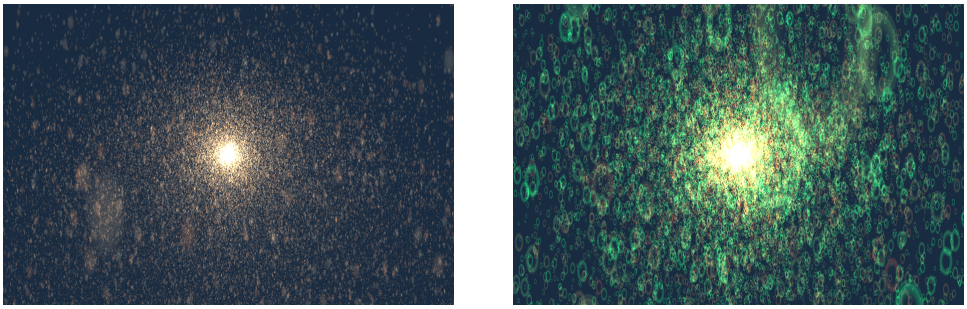
CHAPTER 3

Results

3.1 Running the Prototype

The program can run in any browser that supports WebGPU, if the browser does not support WebGPU, an error message is displayed. The program was tested with Google Chrome Canary 108.0.5357.0 with the *#enable-unsafe-webgpu* flag enabled. The particles are rendered on a canvas, and the GUI allows the user to change various parameters such as the number of particles, their size and their brightness. The user can also control the gravity and the wind. The camera can be rotated horizontally by holding the left mouse button and moving the mouse

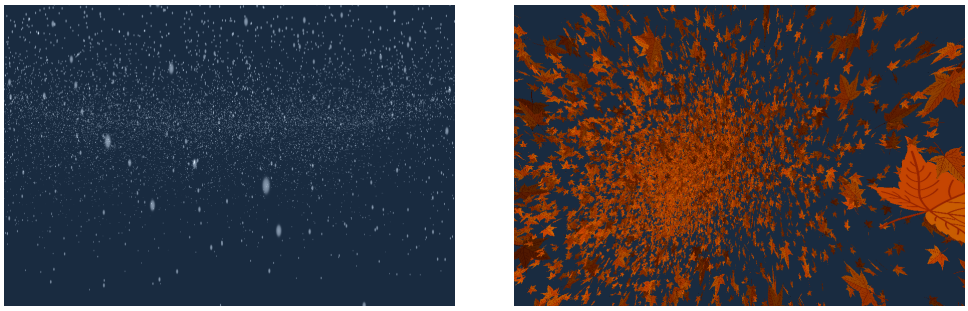
The GUI comes with four presets that showcase different effects created using the three modes. The *Default* preset uses the *default* spawn mode and shows a simple spark effect where particles, which use a circle texture, interpolated colors and additive blending, shoot out of an origin point. The *Circles* preset uses the same mode, but a different sprite and different colors, as well as no gravity, to create a different looking effect. In



(a) The *Default* preset. (b) The *Circles* preset.

Figure 3.1: The particle system running in Chrome Canary.

the *Leaves* preset the particles use a leaf texture without alpha blending, and they spawn in a circle using the *tree* mode. Wind and rotation are also enabled to make the leaves' movement look more natural. The *Snow* preset, using the *snow* mode, creates a simple snow effect where round white particles fall slowly. Under the canvas the frame rate



(a) The *Snow* preset. (b) The *Leaves* preset.

Figure 3.2: The particle system running in Chrome Canary.

is displayed, which can be used to observe the program's performance. For accurate benchmarking however, there is a built-in system that uses timestamp queries.

3.2 Benchmarks

3.2.1 Benchmarking Method

To allow accurate benchmarking, the timestamp query feature is used to retrieve precise timings on the GPU. This feature is not supported by all browsers by default, for creating the following benchmarks Chrome Canary had to be launched with the flag "*–disable-dawn-features=disallow_unsafe_apis*". To use this feature, it has to be requested in the device description, as shown in Section 2.1.1. Then the timestamps can be recorded by using the *GPUCommandEncoder.writeTimestamp* function, which stores them in a GPU buffer[6]. For every frame four timestamps are recorded, two at the beginning and at the end of the compute pass and the render pass respectively. The program records the timestamps of all frames for a specified amount of time (10 seconds by default) and creates a CSV table to store them. By subtracting the beginning of the compute pass from the ending of the render pass, the overall frame time (minus a negligible overhead) can be calculated, as well as the time taken up by the compute pass and render pass. Using the mean values for these time spans, the effect of different parameters on the performance of both the compute pass and the render pass can be observed.

The benchmarking was done on the following system:

In addition to the "*–disable-dawn-features=disallow_unsafe_apis*" flag that the benchmarking system required, the browser was also launched with the"*–disable-gpu-vsync*" flag to ensure that the frame times are not affected by v-sync.

Hardware Specifications	
CPU	AMD Ryzen 5 2600 Six-Core Processor 3.40 GHz
RAM	16GB
GPU	NVIDIA GeForce GTX 1060 6GB
Software	
Operating System	Windows 10 Home 21H1
Browser	Google Chrome Canary 108.0.5359.0

Table 3.1: Specifications of the hardware and software the benchmarks were performed on

3.2.2 Overall Performance

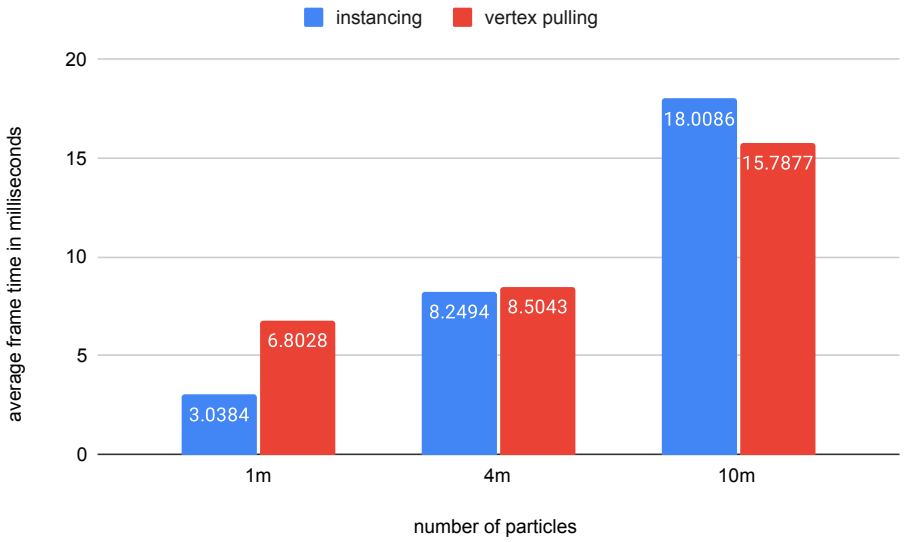


Figure 3.3: The average frame times (in milliseconds) for rendering one million, four million and ten million particles in instancing and vertex pulling modes, using the *Default* preset.

The results of the general benchmark, displayed in Figure 3.3, show that for four million or fewer particles, using instancing for rendering them is more efficient than vertex pulling, but for larger quantities of particles, vertex pulling leads to a shorter rendering time. On a GTX 1060, using vertex pulling, this program can render ten million particles in 15.7877 milliseconds, which is equivalent to a frame rate 63.3 frames per second.

3.2.3 Effect of the number of particles on compute pass time

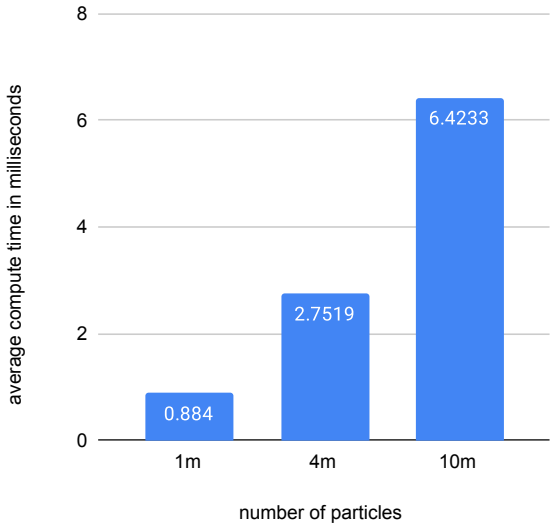


Figure 3.4: The compute pass duration (in milliseconds) for rendering one million, four million and ten million particles on a GTX 1060.

When comparing just the compute pass times of the previous benchmark, as seen in Figure 3.4, the execution time of the compute pass scales roughly linearly with the number of particles simulated.

3.2.4 Effect of particle size on rendering time

For this benchmark, the *usePixelSizes* option was used so that the particles' sizes could be given precisely in pixels. Additive blending was also enabled. The results of the benchmark, shown in Figure 3.5, show that the render time scales better than linearly with the particles' area, i.e., doubling the particle size quadruples the fragments, but rendering time increases by less than 4x..

3.2.5 Comparison of Instancing and Vertex Pulling

Since the prototype allows rendering the particles by using either instancing or vertex pulling, it was possible to compare how those methods differ in performance for rendering a large amount of quads. Two benchmarks were performed, both rendering 10 million particles with a very small size of only 0.1x0.1 pixels (meaning the vast majority of particles do not actually get drawn, making the fragment shader's impact on the render time negligible).

The benchmarks done on the GTX 1060 show that the vertex pulling approach leads to a 41% shorter render pass time. The same benchmark was also performed on the much

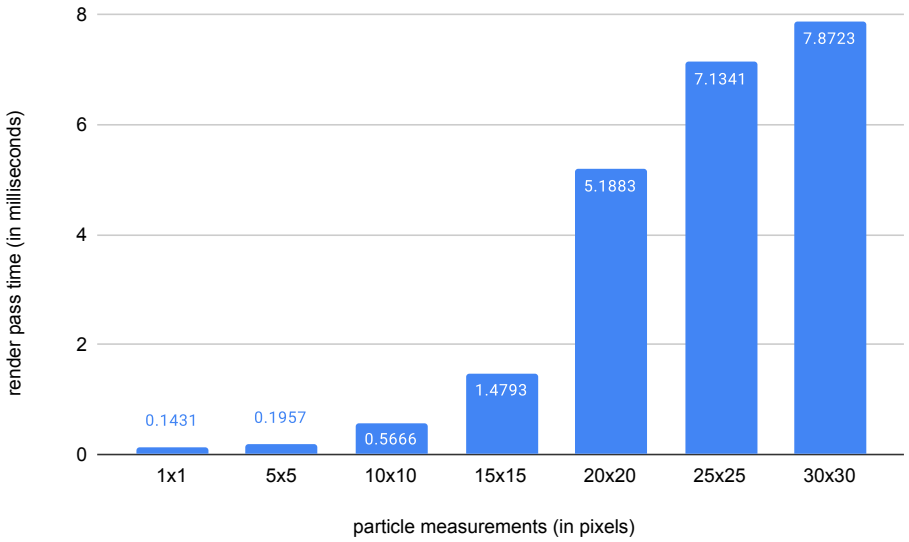


Figure 3.5: The average render pass time for rendering 100,000 particles with different particle sizes on a GTX 1060.

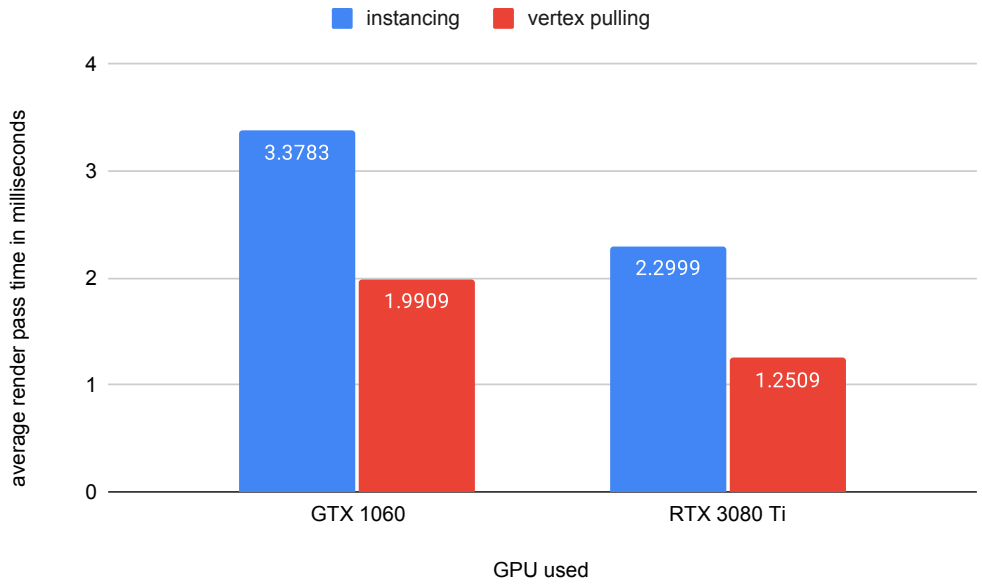


Figure 3.6: The average render pass execution times when rendering 10 million particles of size 0.1x0.1, using instancing and vertex pulling respectively.

more recent Nvidia Geforce RTX 3080 Ti, where rendering using vertex pulling was also significantly faster, with a 45.6% shorter frame time.

3.3 Conclusion and future work

This thesis shows that it is possible to render 10 million particles at a frame rate of roughly 63 frames per second in WebGPU using a compute shader (on a GTX 1060). It also shows that increasing the size of particles has a significant impact on performance, but less than linear with respect to the number of generated fragments. The benchmarks show that using vertex pulling to render the particles leads to significantly better vertex shader performance than using instancing, but that may not be universally true for all GPUs (as shown in a discussion on the WebGPU repository[14]). How the two approaches compare in terms of performance on different devices is a potential future subject of study.

A code repository of the program can be found on GitHub[4].

The particle system prototype is rather basic and could be expanded and modified in various ways. The particles are simulated individually and drawn as textured quads. Another approach could be to calculate the pixel color in the fragment shader rather than sampling a texture, with the differences in performance being a potential subject of further study. Also of interest might be different approaches to achieve similar effects, such as calculating the particle density in a compute shader rather than simulating individual particles, and how these approaches compare in terms of performance and appearance.

Bibliography

[1] Chrome developers: Webgpu. URL: https://developer.chrome.com/docs/web-platform/webgpu/.

[2] Chrome Platform Status. feature: Webgpu. [Accessed 18-Oct-2022]. URL: https://chromestatus.com/feature/6213121689518080.

[3] Kenney • particle pack. [Accessed 18-Oct-2022]. URL: https://kenney.nl/assets/particle-pack.

[4] A particle system in webgpu. [Accessed 28-Nov-2022]. URL: https://github.com/benediktpeter/webgpu-particle-system.

[5] WebGPU. [Accessed 18-Oct-2022]. URL: https://www.w3.org/TR/webgpu/#intro.

[6] WebGPU — w3.org. https://www.w3.org/TR/webgpu/#timestamp. [Accessed 13-Oct-2022].

[7] Webgpu Shading Language. [Accessed 18-Oct-2022]. URL: https://www.w3.org/TR/WGSL/#intro.

[8] WebGPU Shading Language: Compute shaders and workgroups — w3.org. https://www.w3.org/TR/WGSL/#compute-shader-workgroups. [Accessed 13-Oct-2022].

[9] WebGPU Shading Language: Memory layouts — w3.org. https://www.w3.org/TR/WGSL/#memory-layouts. [Accessed 13-Oct-2022].

[10] Free vector: Fallen leaves of different trees vector illustrations set. forest foliage, dry green, yellow, brown, orange leaves isolated on white background. autumn or fall, nature, plants concept for decoration, Dec 2021. URL: https://www.freepik.com/free-vector/fallen-leaves-different-trees-vector-illustrations-set-forest-foliage-dry-gr
21683718.htm#query=leaf&position=4&from_view=keyword.

[11] Shannon Drone. Real-time particle systems on the gpu in dynamic environments. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, page 80–96, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1281500.1281670.

[12] Kim Enarsson. Particle simulation using asynchronous compute: A study of the hardware. 2020. URL: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1439826&dswid=9022.

[13] Austin Eng. Webgpu samples. [Accessed 08-Nov-2022]. URL: https://github.com/austinEng/webgpu-samples.

[14] Gpuweb. Remove vertex input by kainino0x · pull request #411 · gpuweb/gpuweb. URL: https://github.com/gpuweb/gpuweb/pull/411.

[15] JeGX. How to rotate a vertex by a quaternion in glsl (*updated*), Dec 2014. URL: https://www.geeks3d.com/20141201/how-to-rotate-a-vertex-by-a-quaternion-in-glsl/.

[16] Aaron Knoll, R. Keith Morley, Ingo Wald, Nick Leaf, and Peter Messmer. *Efficient Particle Volume Splatting in a Ray Tracer*, pages 533–541. Apress, Berkeley, CA, 2019. doi:10.1007/978-1-4842-4427-2_29.

[17] W. T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, apr 1983. doi:10.1145/357318.357320.