TP01
Particles

Robin Faury

12/15/20

Abstract

In this practical work, we will see how to generate a cloud of particles and how to animate them in real time.

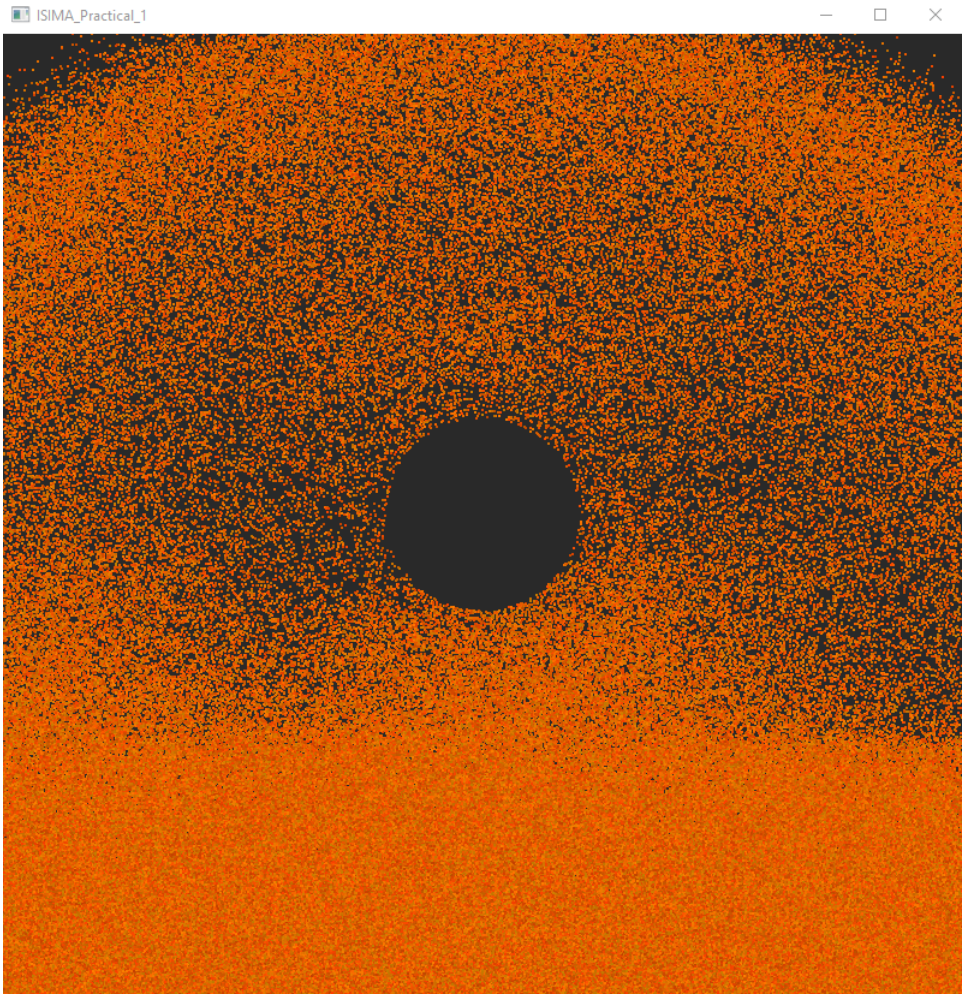

Figure 1: A particle system

1 Introduction

1.1 Cloning the repository

You can find the source of the practical work by cloning this git repository:

```
https://github.com/robinfaurypro/GPGPU_ISIMA_2020-2021.git
```

The CMakeLists file is stored into the Practical_1 folder. Open a terminal on this folder and use thus commands:

```
git submodule update --init
mkdir build
cd build
cmake -G "Visual Studio 15 Win64" ..
```

Of course if you are on UNIX system feel free to use the generator you want. If everything is going well, you can compile and run the executable and get a gray window.

2 Original code

On the main.cpp you can find all the code needed to render the current window. There are two shaders to display particles. The first one is named the vertex shader, it grab a collection of input buffers. The shader will be called for each particles we will create. The second shader is the fragment shader. it display particles and it will be called for each pixel on screen. We can see that the vertex shader can send to the fragment shader some information.
This process is know as the render pipeline. you start with some input buffers, thus buffers will be process by several shader (on per specific purpose) and finally print on a screen or a texture. This pipeline is static and optimized for rendering so it is NOT for a general purpose processing.

Figure 2: The graphic pipeline. Image from render doc application

The shader for general purpose processing is known as the Compute Shader. It only take buffers in inputs and it only release buffer in output. It is also a shader that can read AND write on the same buffer. Other shaders can but only with a very specific setup.

Figure 3: The compute shader. Image from render doc application

The code remainder is quite straightforward:

- Content: the structure to store our data
- Initialization: The initialization function (Compile our shaders)
- MoveParticles: The function to move to particles
- ComputeFrame: The function to call each frame
- Destroy: The function to clean up our data
- main: The main function to run the window

3 GPU information

On the Initialization function you can ask to your device what is its capability. For example, on the main function we check the version and the renderer. you can find here (https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glGetString.xhtml) the list of properties. Check if the shading language version is 430 or more. We need a 430 shading language version to run some compute shaders. We can now ask our GPU his compute capabilities using the glGetIntegeri OpenGL function:

```
int workgroup_count_x;
glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 0, &workgroup_count_x);
// _v for vector (need to be called with 1 and 2 also to get the all vector)
```

Print the value of the vector GL_MAX_COMPUTE_WORK_GROUP_COUNT, GL_MAX_COMPUTE_WORK_GROUP_SIZE and the scalar GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS.

4 Particle buffer

As you can see on the vertex shader, a particle is:

- a 2D position
- a 2D velocity
- a color
- a life time

Create a struct to store of this values in this order and create a vector of Particule in the Content. For your information, glm::vec2 can store a 2D position or velocity. The particles vector can be initialize with 1024*1024 particles. Choose the start state of each particule. For example:

- position = {[0, 1], 0.9}
- velocity = {0, 0}
- color = {1, 0.5, 0}
- life = 100

You can use the std::uniform_real_distribution function with 0 and 1 as parameters (normal distribution) to generate a rand number between 0 and 1.
Now we need to send our vector to the GPU. The GPU work with handle (a kind of pointer). Create a GLuint variable on the Content struct. and call the glGenBuffers.

```
glGenBuffers(1, &content.buffer_);
```

The first parameter is the number of buffer we need and the second is our handle. Add an assert after the call to check the value of content.buffer_. It shouldn't be zero.
OpenGL work as a state machine. We can bind a buffer and all actions we will do will affect the binded buffer. Binding a buffer mean that we are working on it.

```
glBindBuffer(GL_SHADER_STORAGE_BUFFER, content.buffer_);
```

The first action to do is the allocation:

```
glBufferData(
        GL_SHADER_STORAGE_BUFFER,
        content.particles_.size()*sizeof(Particle),
        nullptr,
        GL_STATIC_DRAW);
```

Now we can ask a pointer on the data:

```
void *ptr = glMapBuffer(GL_SHADER_STORAGE_BUFFER, GL_WRITE_ONLY);
```

Adding the WRITE_ONLY flag will help the driver. After the map action you can copy (with memcpy for example) your particle data. After that we must unmap the pointer.

```
void *ptr = glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
```

After this action the ptr variable isn't valid! Finally we can unbind the buffer.

```
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

On the ComputeFrame function uncomment the code. This is the call to the rendering pass. You should see this kind of result:

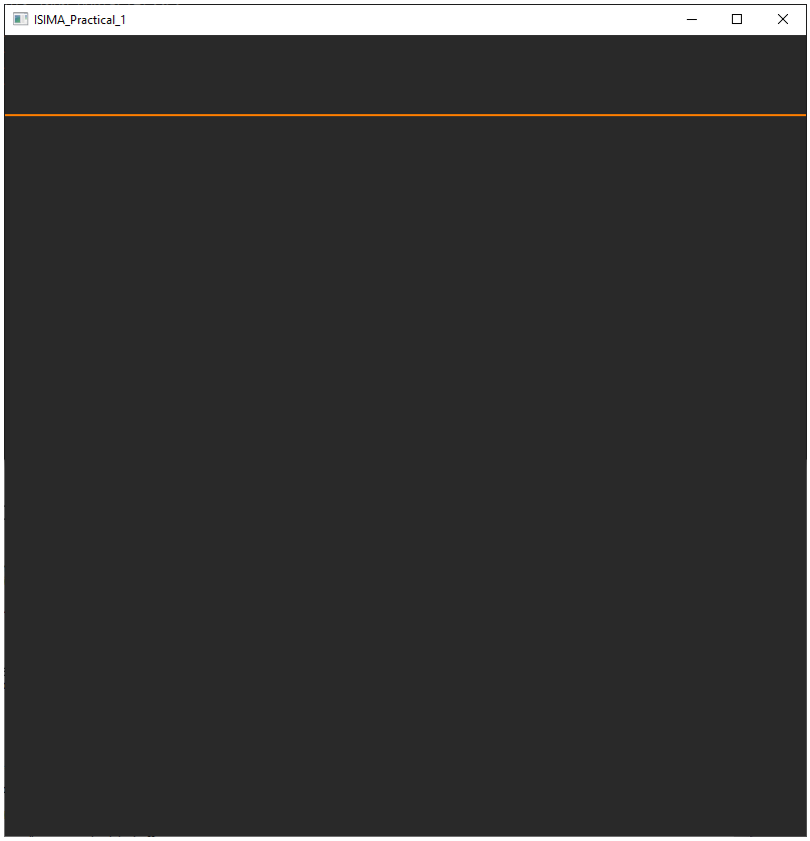

Figure 4: Particles

5 Move particle

5.1 CPU

Firstable we need to write the process of moving the particle in CPU. Most of the time your algorithms must be write in C++ just in case the GPU your user use don't support compute shader. In this case we will use the CPU version as a comparison too. First, grab the GPU (device) buffer on the CPU (host) side using the glMapBuffer function and update the position and velocity of each particles using formulas below.

- Bind the buffer
- Map the buffer using the GL_READ_WRITE option
- reinterpret cast the pointer into a Particle* pointer
- Update each particle
- Unmap the buffer
- Unbind the buffer

$$position_i = position_i + velocity_i * \Delta t + \frac{\Delta t^2 * g}{2}, \tag{1}$$

$$velocity_i = g * \Delta t. \tag{2}$$

With:

- Δt: The time simulated between two frame (0.01 for example)
- g: The gravity {0, -9.8}
- $position_i$: The position of the particule i
- $velocity_i$: The velocity of the particule i

Before modify the buffer, you can check if the buffer uploaded on the GPU is correct (not fill with zeros).

5.2 GPU

5.2.1 Shader

Run the program and take a look at the FPS. A real time application should run at 16 ms per frame. For this 1 048 576 particles simulation we do not have this number. We need to find a way to optimize this. We can imagine using some std::thread but we can use the GPU and its 1000 cores to run 1000 threads.
The first thing to do is to create the compute shader. For that, create a new const char* variable like the vertex shader named kComputeSource. The first line of the shader is the version. After that copy paste your Particle struct in your shader. Now we can define the input buffer and its layout:

```
//...  (Version and Particle struct)
layout(std140, binding=0) buffer Particles {
        Particle particles[];
};
//...
```

The input and output buffer is now named Particles and it is a vector of Particle. You can access to one particle through the variable particles[i].
Next you must specify the number of shader invocation. Remember that the shader can specify how many time it should be run. And because compute shader can compute data in 3D, you can specify a layout of invocation (a thread id can be in 3D).

```
//...
layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;
//...
```

And finally the main function:

```
//...
void main() {
        vec2 position = particles[gl_GlobalInvocationID.x].position_;
        vec2 velocity = particles[gl_GlobalInvocationID.x].velocity_;
        // ... (Move particle)
        particles[gl_GlobalInvocationID.x].position_ = position;
        particles[gl_GlobalInvocationID.x].velocity_ = velocity;
}
//...
```

Remember that gl_GlobalInvocationID is the id of the thread so it's like the index of the particle in the vector. You can re-implement your algorithm in this main program.

5.2.2 Program

We have now a shader code. We need to compile it and send it to the GPU. We need to make the same process as we did for the vertex shader.

- Create a shader but with the GL_COMPUTE_SHADER flag
- Send the source
- Compile the source
- Check errors

Our shader must be wrapped inside a program. Add a Handle (GLuint) in the content struct to store the program name (kernel_move_particles_ for example). The link of your program should return a valid handle. You can delete the shader, you just need the program.

5.2.3 Dispatch

We can now use the new compute program to move particles. In the MoveParticles function deactivate the CPU code (using #if 0 or comment or a const bool) and use the compute program:

```
glUseProgram(content.kernel_move_particles_);
```

As usual, you need to bind the particle buffer. But this time we also need to specify the binding location. On your shader you set binding=0:

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, content.buffer_);
```

And finally you can run the dispatch function:

```
glDispatchCompute(content.particles_.size()/128, 1, 1);
```

We set the number of particles divide by 128 because the layout value on x on the shader is 128.
The dispatch function will run the simulation asynchronous. To be sure to right measure the duration of the computation we need to add a synchronization point (a barrier). It is also very important to add a barrier if you dump data from the GPU just after a dispatch. You can grab just half of your buffer.

```
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

After that we can clean the OpenGL state machine by disable the program:

```
glUseProgram(0);
```

5.2.4 Destroy

When the application is killed, we must clean our GPU resources. On the destroy function you need to delete the two program and the buffer:

```
glDeleteProgram(content.kernel_draw_particles_);
glDeleteProgram(content.kernel_move_particles_);
glDeleteBuffers(1, &content.buffer_);
```

5.2.5 Results

If everything is ok you should have this kind of result:

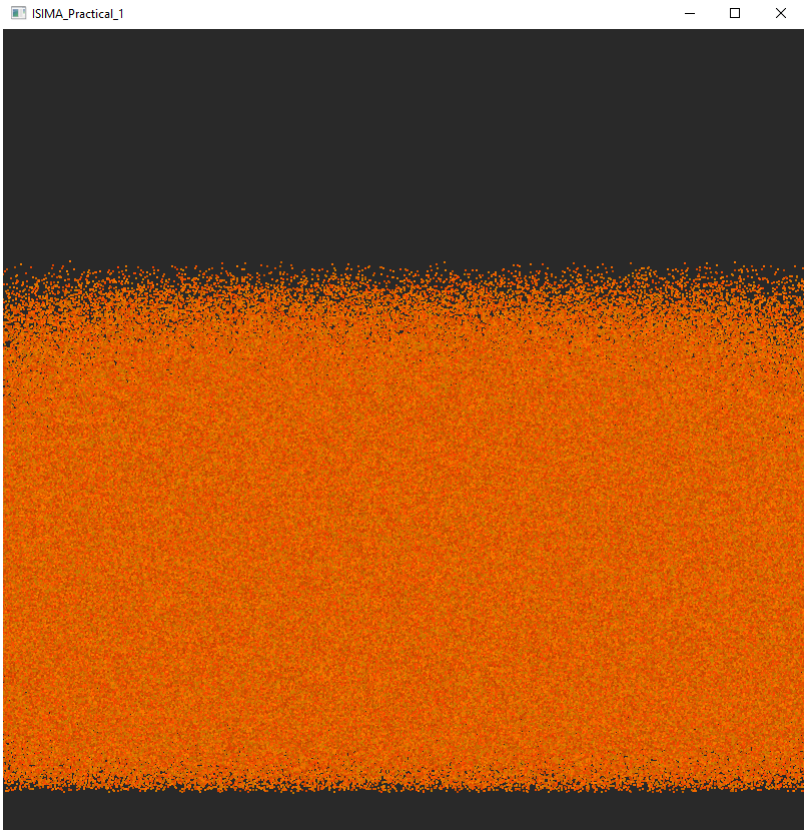

Figure 5: Particles falling

For this picture I set a random orange color for the color and a random velocity.

6 Bonus

We just simulate a atmosphere free domain. We can add a limit of the velocity on the shader.

```
velocity = clamp(velocity, -5.f, 5.f);
```

We saw that GPU doesn't have random function. But, we can simulate random function using sin function and magic number. On your compute shader add the random function above the main:

```
float random(vec2 st) {
        return fract(sin(dot(st.xy, vec2(12.9898, 78.233)))*43758.5453123);
}
```

With this random function we can simulate a very rough ground. For that simply get a rand value from the position when the position is above 0 and compute the new velocity.

```
if (position.y < 0.0) {
        float rand = random(position);
        position.y = 0.0;
        velocity = -velocity;
        vec2 normal = vec2(rand*2.0-1.0, 1.0);
        normal = normalize(normal);
        velocity = reflect(velocity, normal)*0.8;
}
```

You can also add a circle on the middle of the screen. If the position less the center of the circle (0.5, 0.5) is less than the radius, then reflect the velocity with the normal of the sphere and recompute the position with the new velocity.

7 Uniform value

Some additional value can be add to our shader. It is called uniform. For example you can define:

```
uniform float time;
```

and have access to the time variable in your shader code. you can set the time value on the CPU side after the glUseProgram:

```
glUniform1f(glGetUniformLocation(program_, "time"), glfwGetTime());
```

The time variable can be used to move the center of the sphere (using a sinus to keep the value between 0 and 1) for example.