Advanced Graphics Programming

Compute Shaders

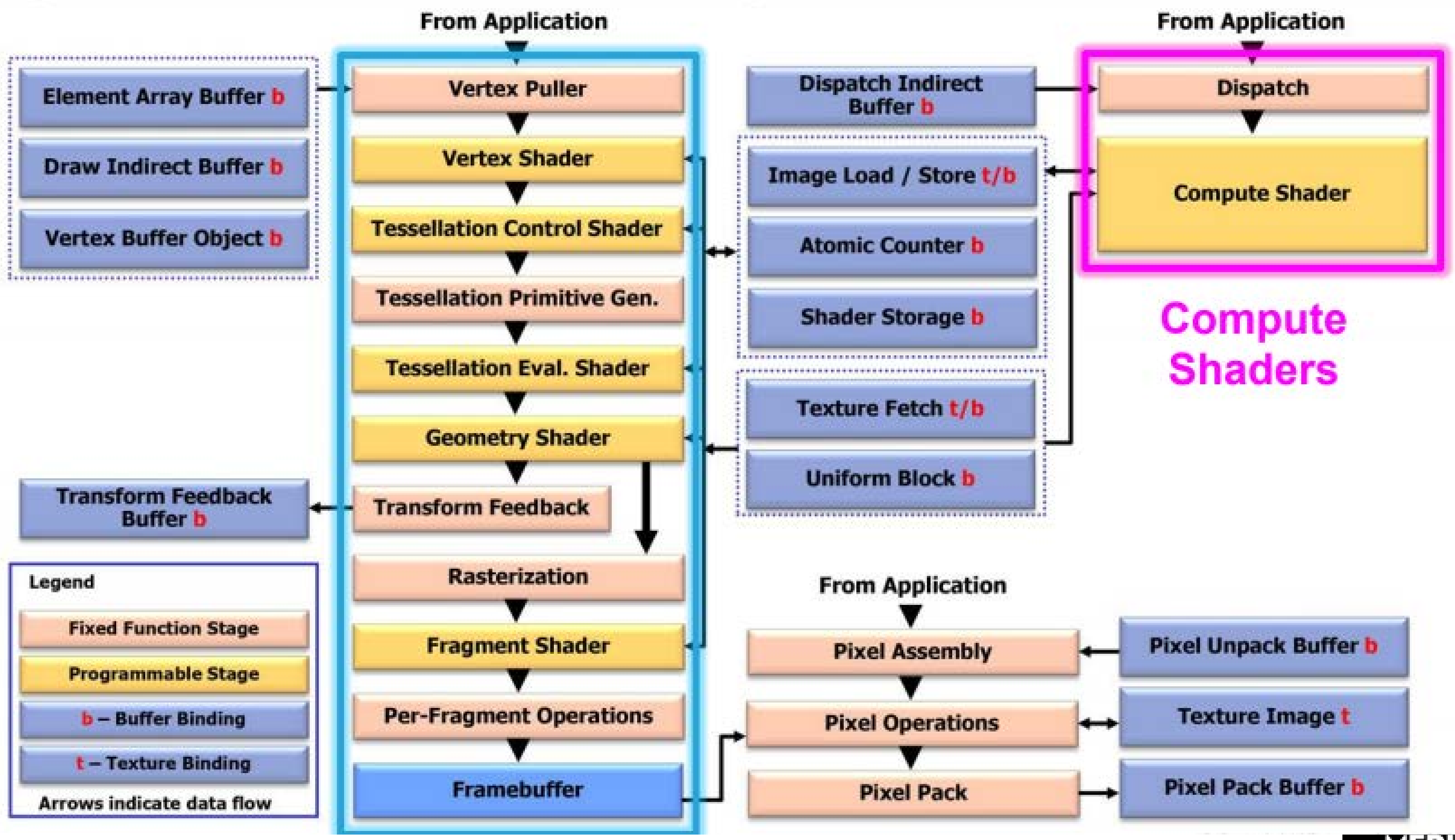
MEDIA DESIGN SCHOOL GAME DEV

Overview

- Compute shaders are a way to take advantage of the enormous computational power of graphics processors that implement OpenGL.
- Just like all shaders in OpenGL, they are written in GLSL
- Run in large parallel groups that simultaneously work on huge amounts of data
- In addition to the facilities available to other shaders such as texturing, storage buffers, compute shaders are able to synchronize with one another and share data among themselves to make general computation easier
- They stand apart from the rest of the OpenGL pipeline and are designed to provide as much flexibility to the application developer as possible

MEDIA DESIGN SCHOOL GAME DEV

OpenGL 4.3 with Compute Shaders



Overview

- Obviously, graphics benefit from this GPU architecture, as the architecture was designed for graphics.
- However, some non-graphical applications benefit from the massive amount of computational power a GPU can provide with its parallel architecture.
- Using the GPU for non-graphical applications is called **general purpose GPU (GPGPU) programming**
- To take advantage of the parallel architecture of the GPU, we need a large amount of data elements that will have similar operations performed on them so that the elements can be processed in parallel

Overview

- For example
  - shading pixels is a good example, as each pixel fragment being drawn is operated on by the pixel shader
  - Particle systems provide yet another example, where the physics of each particle can be computed independently provided we take the simplification that the particles are all behave same.

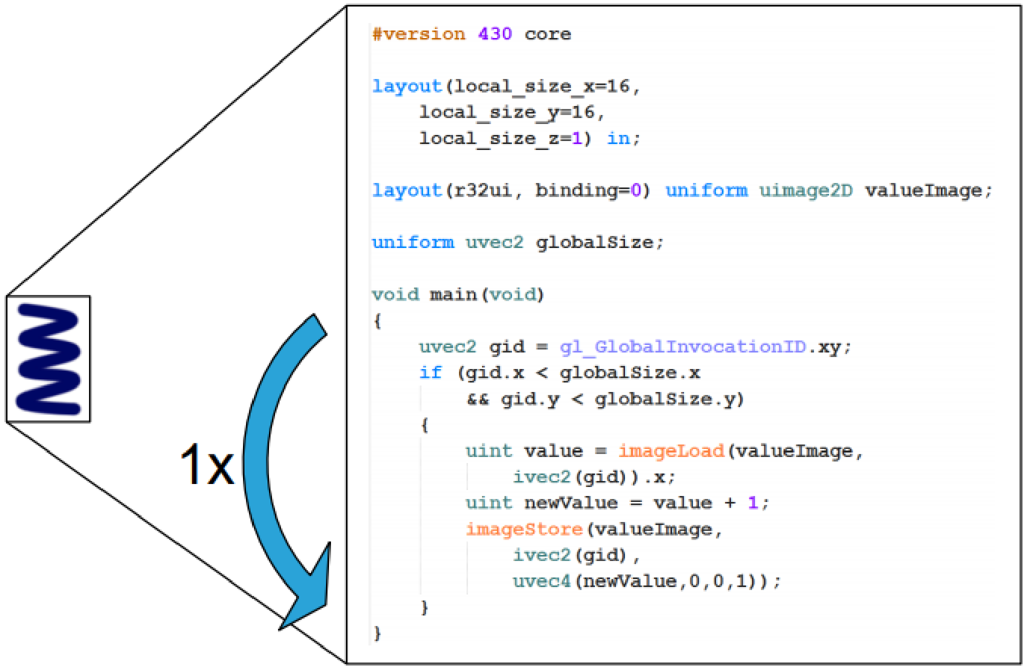
MEDIA DESIGN SCHOOL GAME DEV

Overview

- Nvidia GTX 1080Ti
- 28 Streaming Multiprocessors (SM)
- Each SM has 128 Cores
- Therefore 3584 CUDA Cores
- Maximum number of threads in flight at a time = 2048 x No. of SMs
- So 2048 x 28 = 57344 Threads

Threads

- Execution of the Compute Shader happens per thread per core.
- So a 1 thread = 1 invocation of the shader.

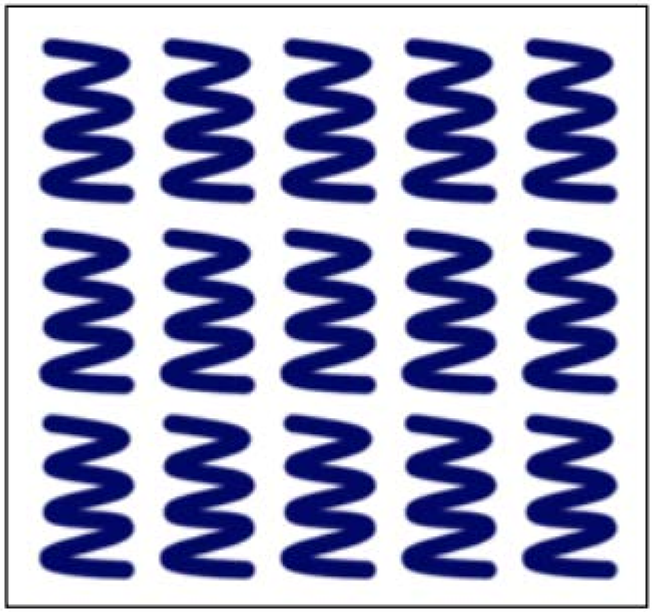


```
#version 430 core

layout(local_size_x=16,
    local_size_y=16,
    local_size_z=1) in;

layout(r32ui, binding=0) uniform uimage2D valueImage;

uniform uvec2 globalSize;

void main(void)
{
    uvec2 gid = gl_GlobalInvocationID.xy;
    if (gid.x < globalSize.x
        && gid.y < globalSize.y)
    {
        uint value = imageLoad(valueImage,
            ivec2(gid)).x;
        uint newValue = value + 1;
        imageStore(valueImage,
            ivec2(gid),
            uvec4(newValue,0,0,1));
    }
}
```

Work Groups

- In GPU programming, the number of threads desired for execution is divided up into a grid of **Work groups**.
- A work group is executed on a single multiprocessor.
- Could be 1D, 2D or 3D.
- Specified in shader code

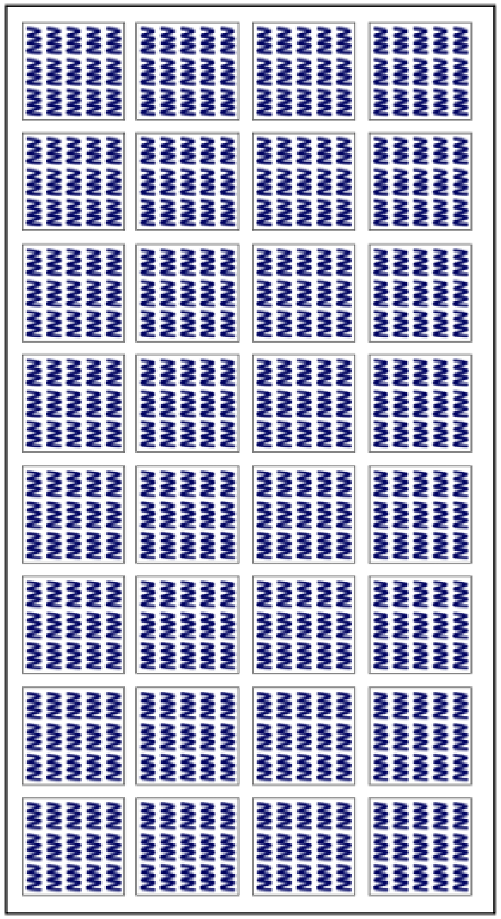
```
layout(
   local_size_x = 5,
   local_size_y = 3,
   local_size_z = 1
) in;
```



Dispatch Groups

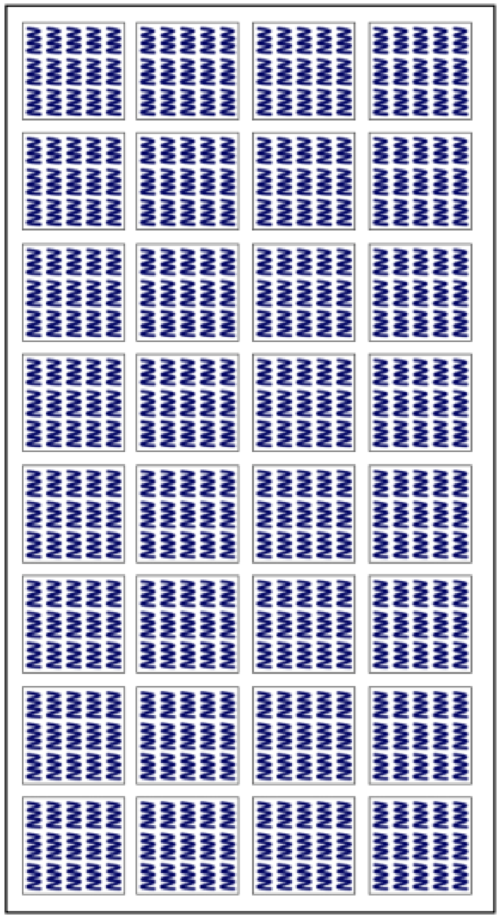
- A grid of Works Groups is Specified in Displatch.
- This also could be in 1D, 2D or 3D.
- The Size is specified in an OpenGL call.

```
glDispatchCompute(
   4 /* x */,
   8 /* y */,
   1 /* z */
);
```



Dispatch Work Group

- The total number of invocations of the compute shader will be the size of this array times the size of the local workgroup declared in the shader code.
- This can produce an extremely large amount of work for the graphics processor and it is relatively easy to achieve parallelism using compute shaders.



Thread Location

- To know where in the local workgroup you are and where that workgroup is within the larger global workgroup.
- ***gl_NumWorkGroups* (vec3)** -
  - Contains the number of work groups passed to dispatch function.
  - **glDispatchCompute()** (*num_groups_x*, *num_groups_y*, and *num_groups_z*).
- ***gl_WorkGroupSize (vec3)***
  - Size of the local workgroup as declared by the local_size_x, local_size_y and local_size_z.
  - **layout qualifiers in the shader**.

MEDIA DESIGN SCHOOL GAME DEV

Thread Location

- ***gl_WorkGroupID (vec3)***
  - Is the location of the current local workgroup within the larger global workgroup.
- ***gl_LocalInvocationID (vec3)***
  - Current invocation of a compute shader within the local workgroup.
- ***gl_LocalInvocationIndex (int)***
  - is a flattened form of gl_LocalInvocationID.
- ***gl_GlobalInvocationID (vec3)***
  - This value **uniquely identifies this particular invocation** of the compute shader among *all* invocations of this compute dispatch call.
  - **It is used to store and retrieve data of the work done by a particular invocation.**
  - Its exact value = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID.

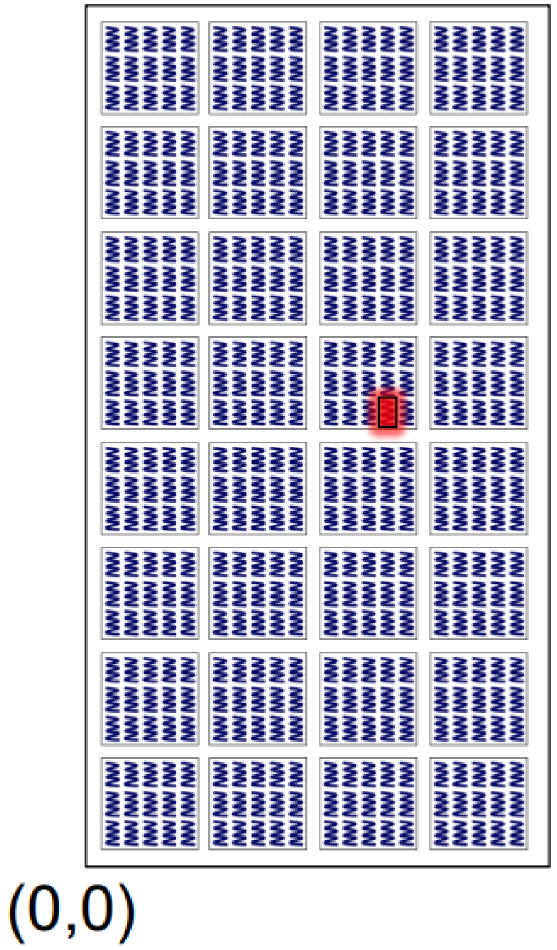
Thread Location

- Built in GLSL variables to get the location of a thread.

`gl_WorkGroupID`
is `(2,4,0)`

`gl_LocalInvocationID`
is `(3,0,0)`

`gl_GlobalInvocationID`
is `(13,12,0)`



Work Group Size Limits

Limits on work group size per dimension:

```
int dim = 0; /* 0=x, 1=y, 2=z */
int maxSizeX;
glGetIntegeri_v(
   GL_MAX_COMPUTE_WORK_GROUP_SIZE,
   dim, &maxSizeX);
```

Limit on total number of threads per work group:

```
int maxInvoc;
glGetIntegeri(
   GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS,
   &maxInvoc);
```

Compute Example

- If you had a GPU with sixteen multiprocessors, you would want to break up your problem into at least sixteen work groups so that each multiprocessor has work to do.
- The hardware actually divides these threads up into *warps* (thirty-two threads per warp), as each CUDA core processes a thread
- “Fermi” multiprocessor has thirty-two CUDA cores.
- For performance reasons thread groups should be in multiples of warp size
- The work group needs to be split to balance work lo to each core.

Comparison

NVIDIA

- Warp size
- 32 threads per warp
- Recommends thread group size to be in multiples of 32

ATI

- “Wavefront” size
- 64 threads per wavefront
- Recommends thread group size to be in multiples of 64

Compute Example

- For example if you are working on a 1280 x 720 image and the local work group size is 16 x16,
- As specified in the compute shader.

```
Layout (local_size_x = 16,
        local_size_y = 16 ,
        local_size_z =  1) in;
```

- And In the render function you would call

```
glDispatchCompute(1280/16, 720/16, 1);
```

Compute Example

```
#version 430 core
layout (local_size_x = 16, local_size_y = 16) in;

// An image to store data into.
layout (rg32f) uniform image2D image;
void main(void) {
        // Store the local invocation ID into the image.
        imageStore(image, //- image
        ivec2(gl_GlobalInvocationID.xy), //- pixel location
        vec4(vec2(gl_LocalInvocationID.xy) /
        vec2(gl_WorkGroupSize.xy),
        0.0, 0.0));// current location in local work group
}
```

Image Store

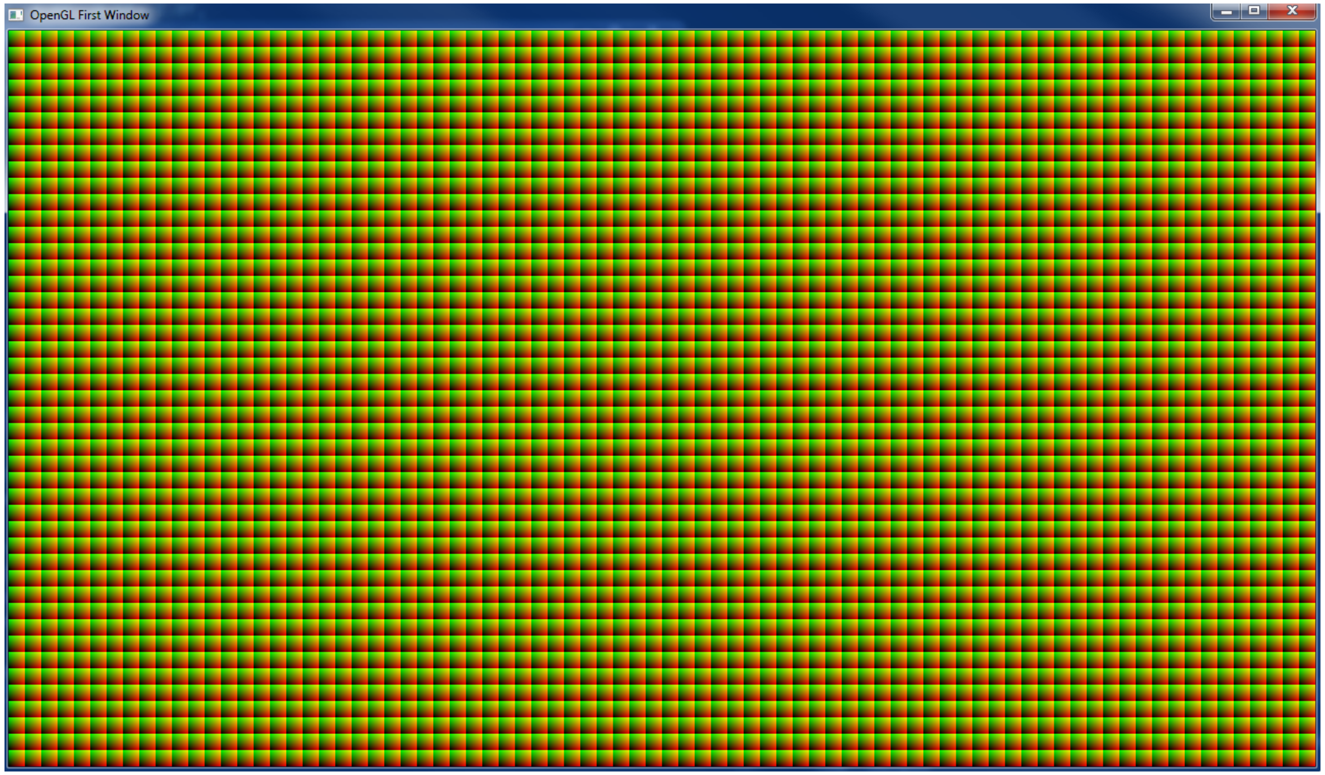
- imageStore : write a single texel into an image

```
void imageStore( gimage2D image,
                 ivec2 P,
                 gvec4 data);
```

image: Specify the image unit into which to store a texel.
P: Specify the coordinate at which to store the texel.
data: Specifies the data to store into the image.

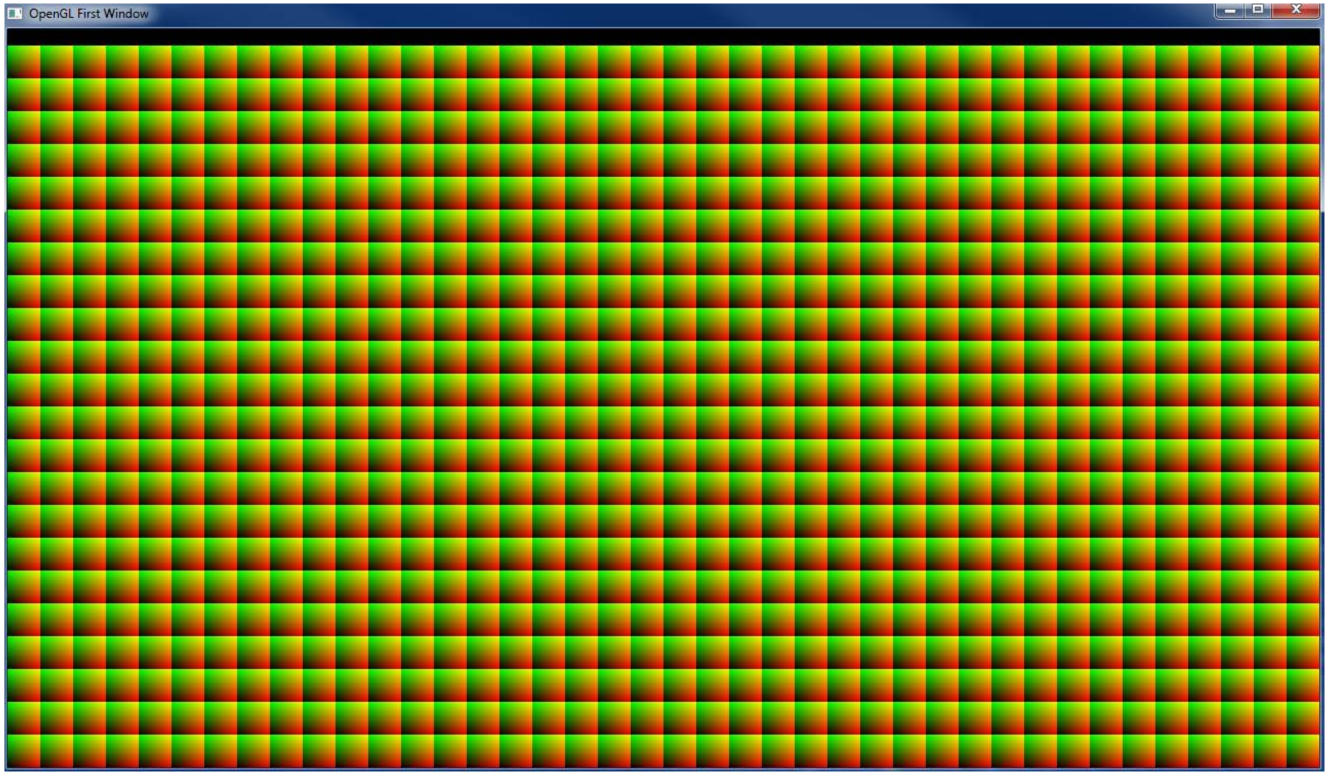
Output

- glDispatchCompute(1280/16, 720/16, 1);
- layout (local_size_x = 16, local_size_y = 16)



Output

- glDispatchCompute(1280/32, 720/32, 1);
- layout (local_size_x = 32, local_size_y = 32)



Synchronization

- The graphics processor will run the work in parallel and the invocations that execute the compute shader can be considered to be a team trying to accomplish a task.
- Cooperation between the invocations is enabled by allowing them to communicate via *shared* variables.
- Furthermore, it is possible to sync up all the invocations in the local workgroup so that they reach the same part of your shader at the same time.

MEDIA DESIGN SCHOOL GAME DEV

Synchronization

- If no communication between the invocations is required and they can all run completely independently of each other, then this likely isn't going to be an issue.
- However, if the invocations need to communicate with each other either through images and buffers or through shared variables, then it may be necessary to synchronize their operations with each other.
- There are two types of synchronization commands.
- The first is an execution barrier, which is invoked using the **barrier()** function. Similar to Tessellation control shader.
- When an invocation of a compute shader reaches a call to **barrier()**, it will stop executing and wait for all other invocations within the same local workgroup to catch up

Synchronization

- The second type of synchronization primitive is the memory barrier. The heaviest, most brute-force version of the memory barrier is **memoryBarrier()**.
- When **memoryBarrier()** is called, it ensures that any writes to memory that have been performed by the shader invocation have been committed to memory rather than lingering in caches or being scheduled after the call to **memoryBarrier()**
- **void glMemoryBarrier(GLbitfield barriers);**
- There are various bits for different operations like buffer access, image access, etc.
- Specify **GL_ALL_BARRIER_BITS** for barriers.

Example Project

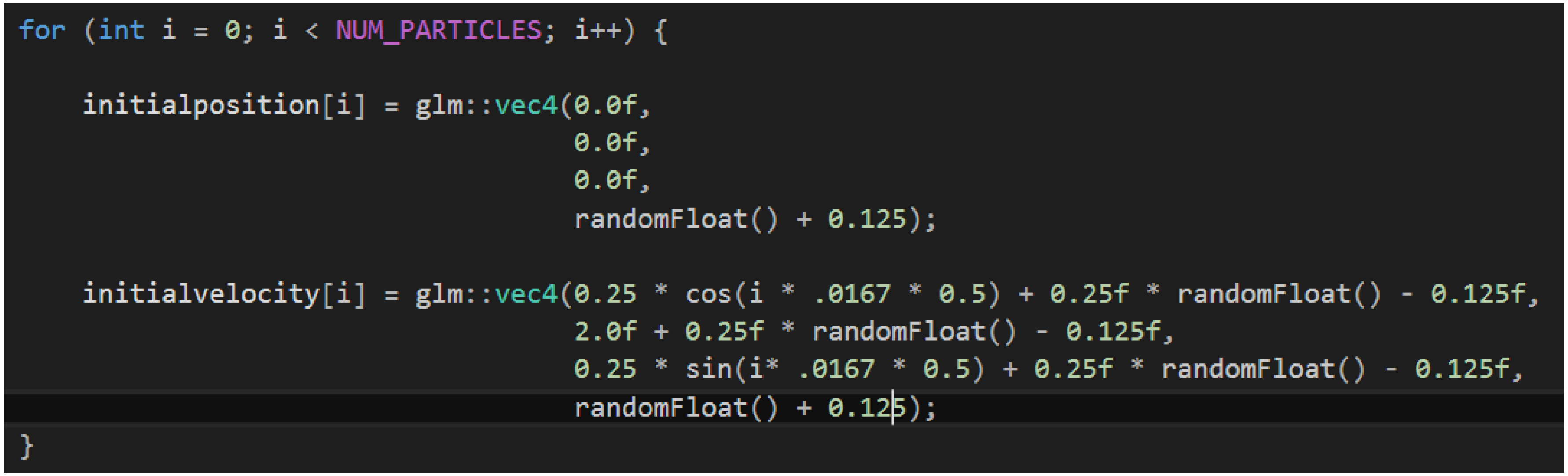
Example project

- Port the Fountain Particle System to GPU.
- Have 2 vec4 vectors to store initial values of position and velocity.
- Create buffer objects and renderPrograms

- std::vector<glm::vec4> initialposition;
- std::vector<glm::vec4> initialvelocity;
- GLuint posVbo, velVbo, initVelVbo, particleVao;
- GLuint computeProgram, renderProgram;

Example Project – Init()

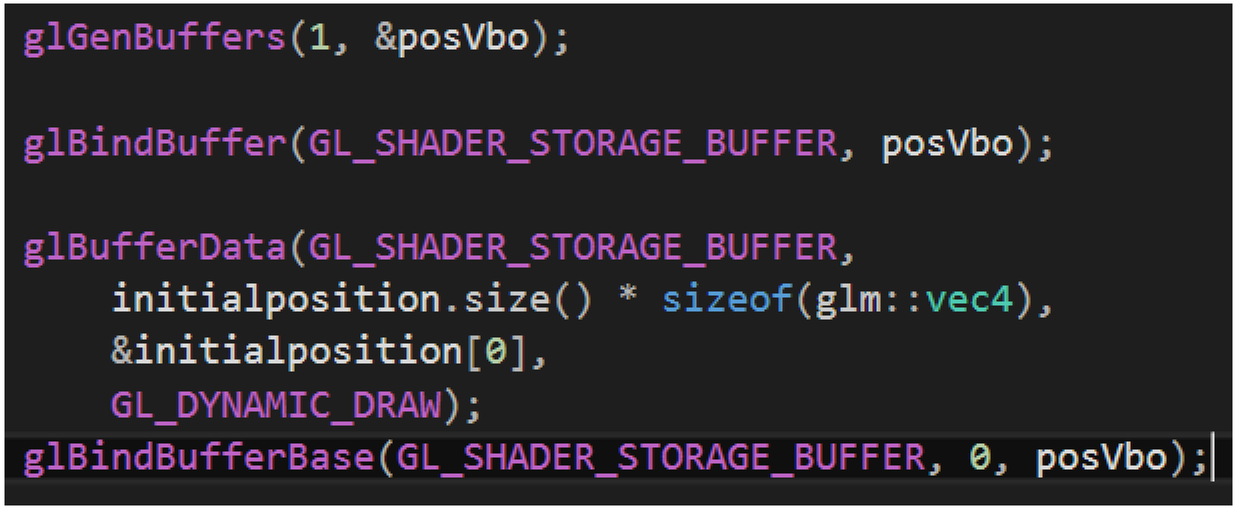
- Store initial values
- #define NUM_PARTICLES 128 * 20000



```
for (int i = 0; i < NUM_PARTICLES; i++) {

    initialposition[i] = glm::vec4(0.0f,
                                   0.0f,
                                   0.0f,
                                   randomFloat() + 0.125);

    initialvelocity[i] = glm::vec4(0.25 * cos(i * .0167 * 0.5) + 0.25f * randomFloat() - 0.125f,
                                   2.0f + 0.25f * randomFloat() - 0.125f,
                                   0.25 * sin(i* .0167 * 0.5) + 0.25f * randomFloat() - 0.125f,
                                   randomFloat() + 0.125);
}
```

Example Project – Init()

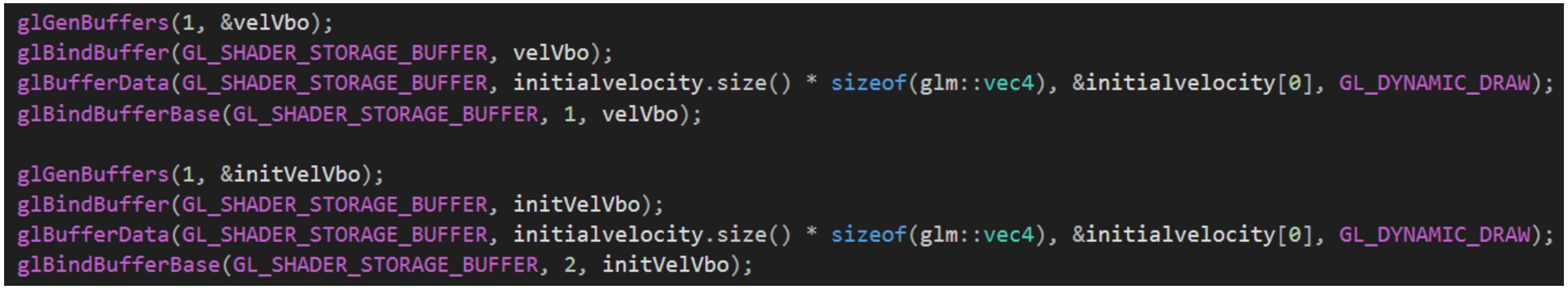
- Use storage Buffer Object - is a Buffer Object that is used to store and retrieve data from within the OpenGL Shading Language.
- Store position information.



```
glGenBuffers(1, &posVbo);

glBindBuffer(GL_SHADER_STORAGE_BUFFER, posVbo);

glBufferData(GL_SHADER_STORAGE_BUFFER,
    initialposition.size() * sizeof(glm::vec4),
    &initialposition[0],
    GL_DYNAMIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, posVbo);
```

Example Project – Init()

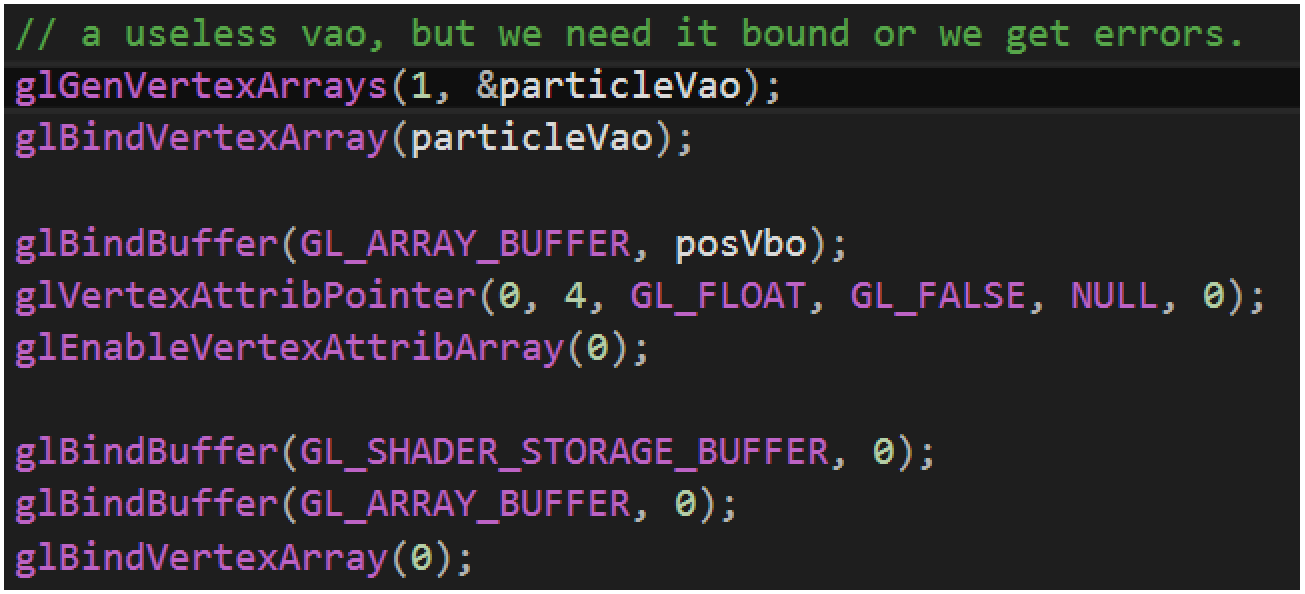
- Store velocity information
- The other buffer stores initial velocity values.



```
glGenBuffers(1, &velVbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, velVbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, initialvelocity.size() * sizeof(glm::vec4), &initialvelocity[0], GL_DYNAMIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, velVbo);

glGenBuffers(1, &initVelVbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, initVelVbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, initialvelocity.size() * sizeof(glm::vec4), &initialvelocity[0], GL_DYNAMIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, initVelVbo);
```

Example Project – Init()

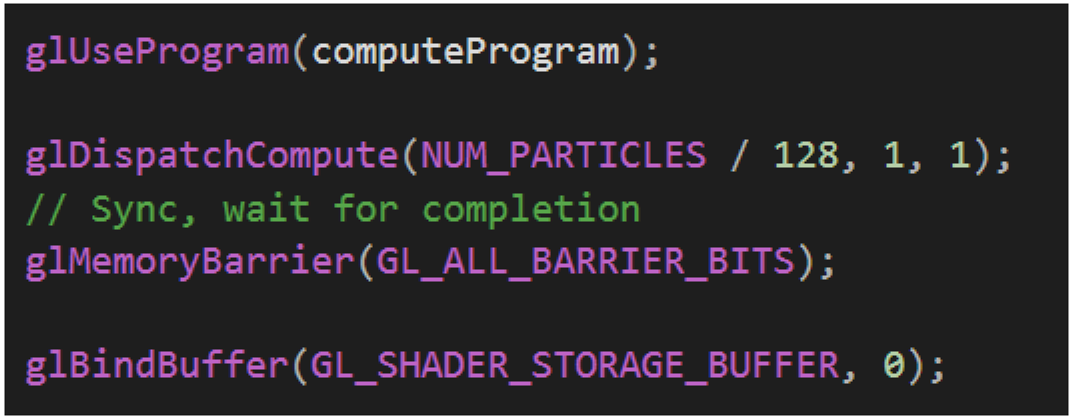
- Create a VAO which doesn't get used by if not included the program will not work.
- Then set only the position attribute as this will be required while rendering.



```
// a useless vao, but we need it bound or we get errors.
glGenVertexArrays(1, &particleVao);
glBindVertexArray(particleVao);

glBindBuffer(GL_ARRAY_BUFFER, posVbo);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, NULL, 0);
glEnableVertexAttribArray(0);

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

Example Project – Render()

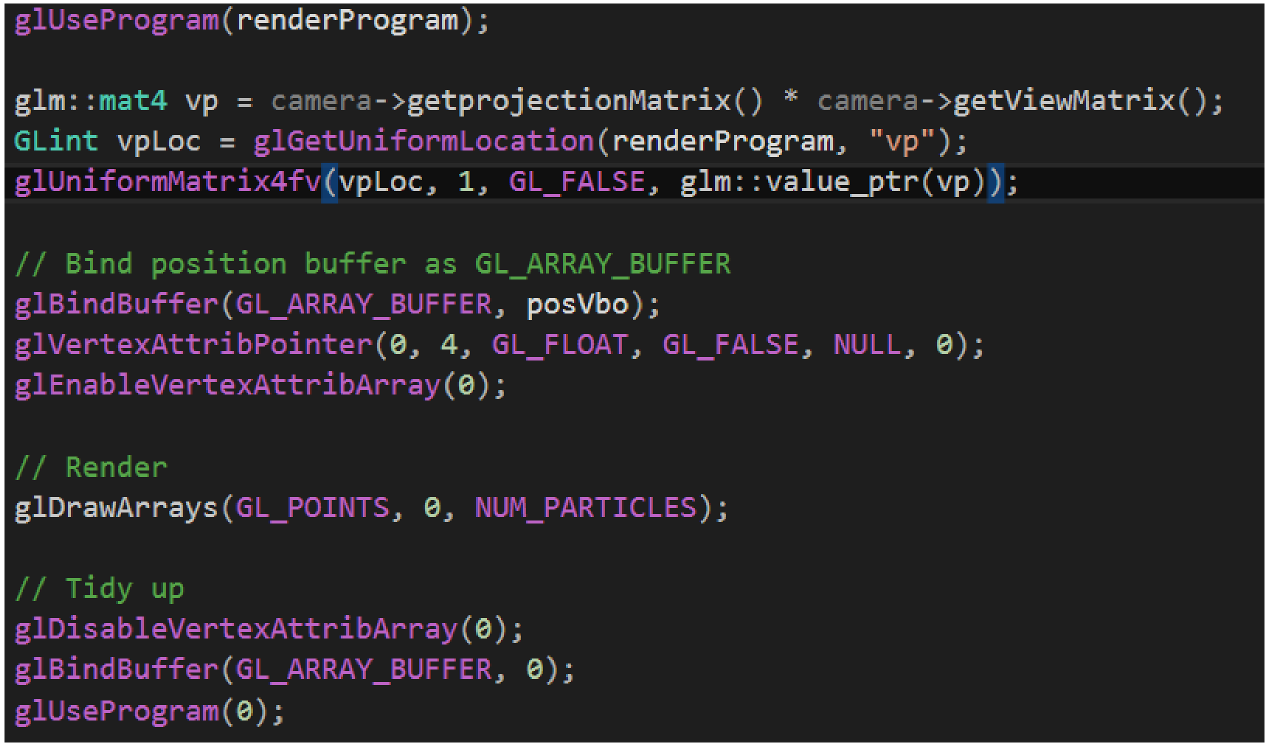
- In the render function
- Set the compute program first
- Call Dispatch the set the work group size
- And call memory barrier so that position of all particles are calculated



```
glUseProgram(computeProgram);

glDispatchCompute(NUM_PARTICLES / 128, 1, 1);
// Sync, wait for completion
glMemoryBarrier(GL_ALL_BARRIER_BITS);

glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

Example Project – Render()

- Then set render program and render the particles.
- Note the VBO is bound instead of VAO



```
glUseProgram(renderProgram);

glm::mat4 vp = camera->getprojectionMatrix() * camera->getViewMatrix();
GLint vpLoc = glGetUniformLocation(renderProgram, "vp");
glUniformMatrix4fv(vpLoc, 1, GL_FALSE, glm::value_ptr(vp));

// Bind position buffer as GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, posVbo);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, NULL, 0);
glEnableVertexAttribArray(0);

// Render
glDrawArrays(GL_POINTS, 0, NUM_PARTICLES);

// Tidy up
glDisableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glUseProgram(0);
```

Compute Shader

- In the compute shader
- Set the core version `#version 440 core`
- Set the local work group size

```
#define WORK_GROUP_SIZE 128

layout( local_size_x = WORK_GROUP_SIZE, local_size_y = 1, local_size_z = 1 ) in;
```

- Retieve the 3 vectors passed in

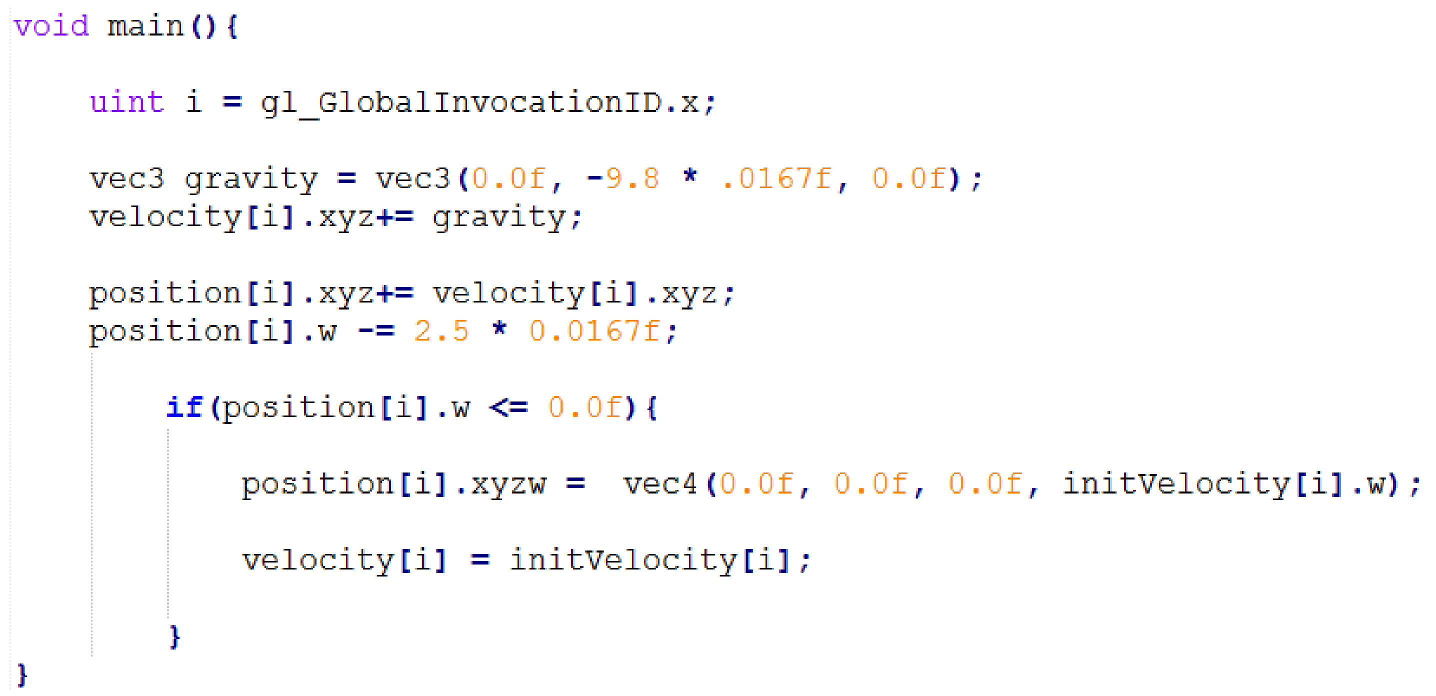
```
layout(std430, binding = 0) buffer positionBuffer{vec4 position[]; };
layout(std430, binding = 1) buffer velocityBuffer{vec4 velocity[]; };
layout(std430, binding = 2) buffer initVelocityBuffer{vec4 initVelocity[]; };
```

Compute Shader

- Std430 is a memory layout qualifier.
- There are other like packed, shared, std140.
- Packed: This layout type means that the implementation determines everything about how the fields are laid out in the block.
- Shared: All variables are active and not optimized.
- Std 140: Data is not packed and floats are not equivalent to C/ C++. So use it for Int data instead of floats.

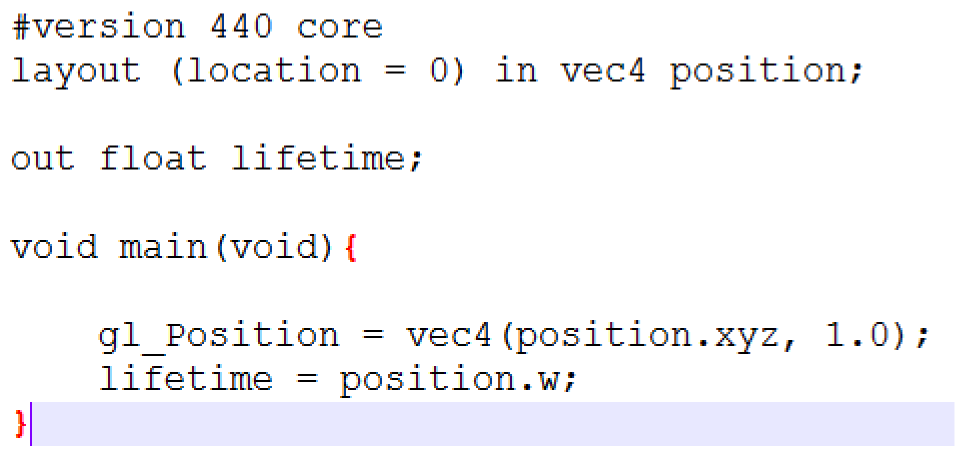
Compute Shader

- For each invocation id calculate the new position based on velocity and update lifetime.
- If lifetime is less zero set position and velocity to default values



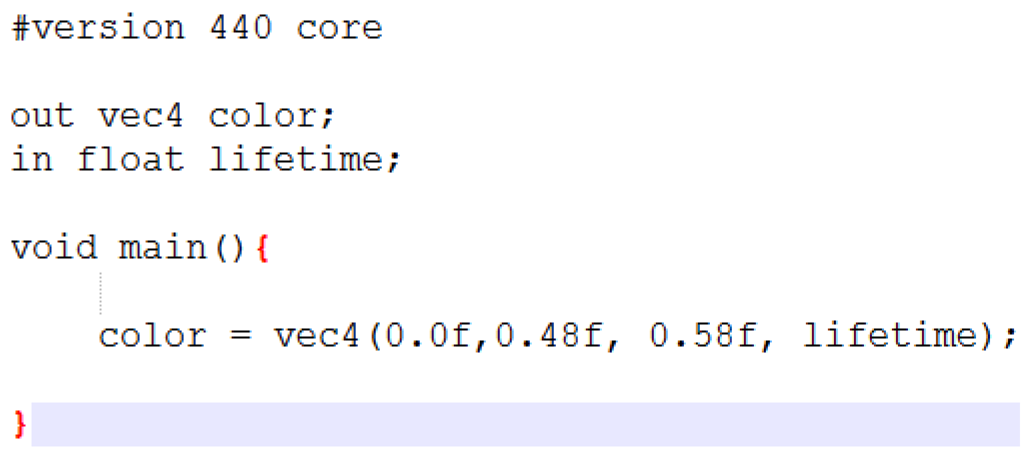
```
void main(){

    uint i = gl_GlobalInvocationID.x;

    vec3 gravity = vec3(0.0f, -9.8 * .0167f, 0.0f);
    velocity[i].xyz+= gravity;

    position[i].xyz+= velocity[i].xyz;
    position[i].w -= 2.5 * 0.0167f;

        if(position[i].w <= 0.0f){

            position[i].xyzw =  vec4(0.0f, 0.0f, 0.0f, initVelocity[i].w);

            velocity[i] = initVelocity[i];

        }
}
```

Vertex and Fragment Shader



```
#version 440 core
layout (location = 0) in vec4 position;

out float lifetime;

void main(void){

    gl_Position = vec4(position.xyz, 1.0);
    lifetime = position.w;
}
```

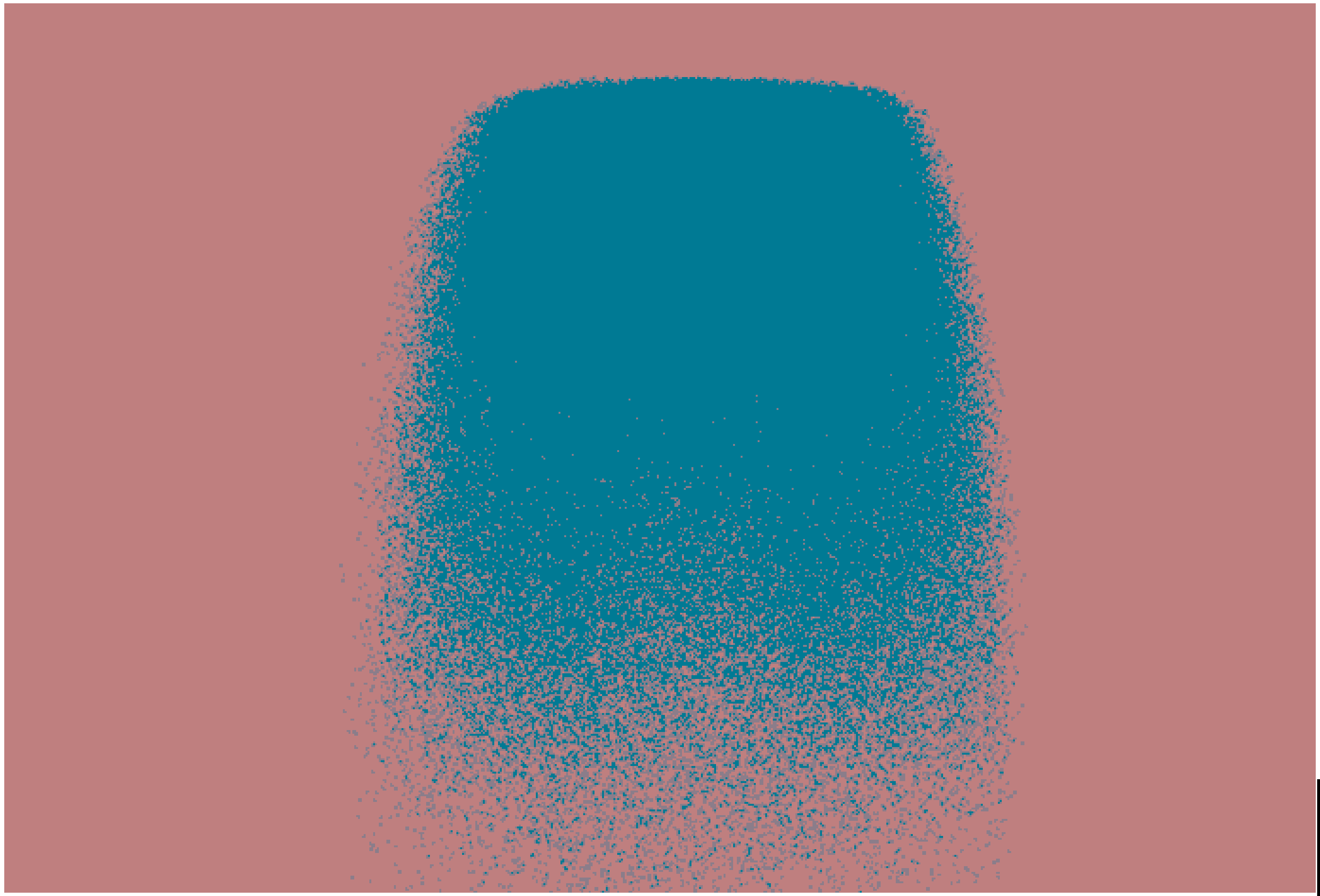
Vertex Shader



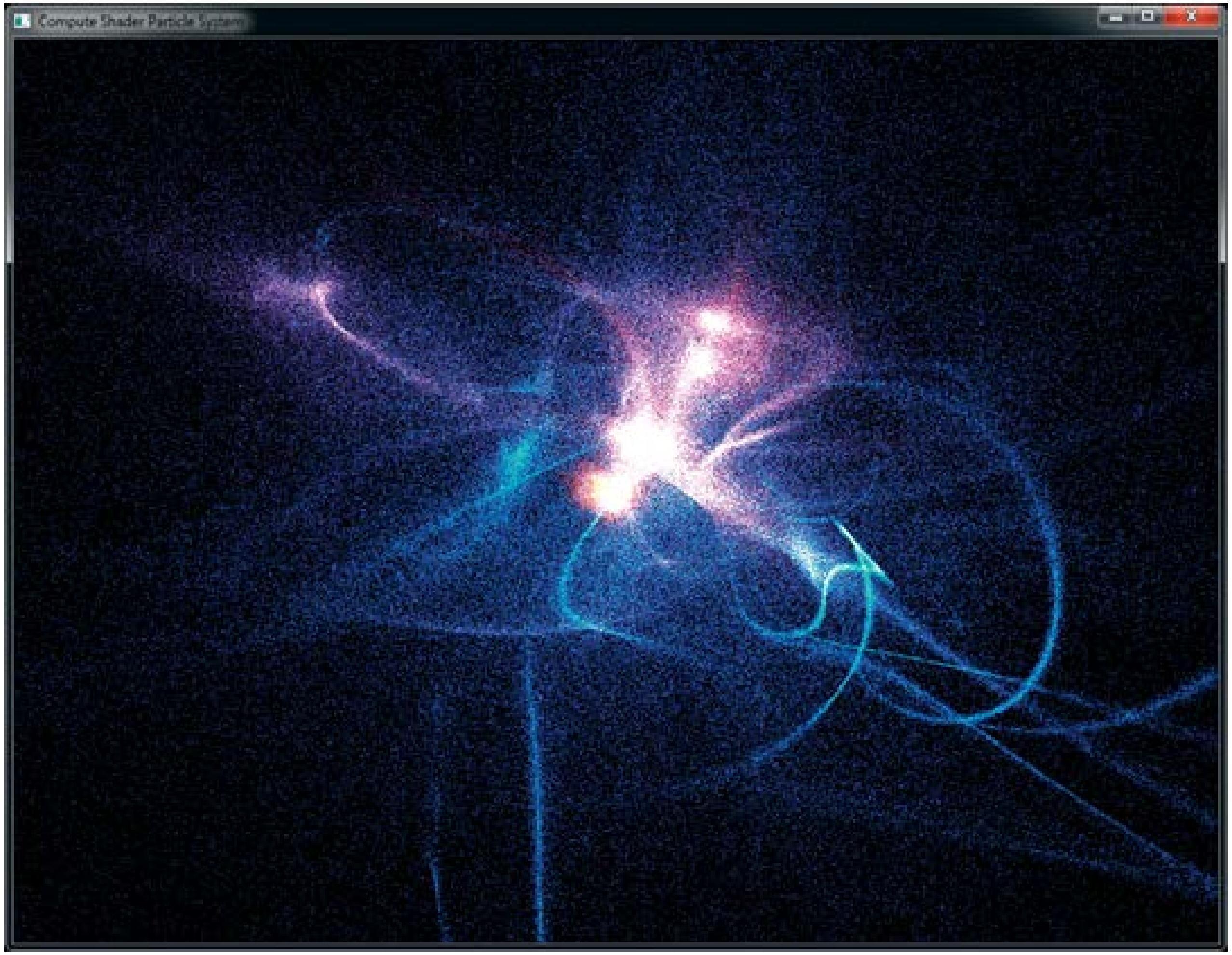
```
#version 440 core

out vec4 color;
in float lifetime;

void main(){

    color = vec4(0.0f,0.48f, 0.58f, lifetime);

}
```

Fragment Shader

Output



Usage

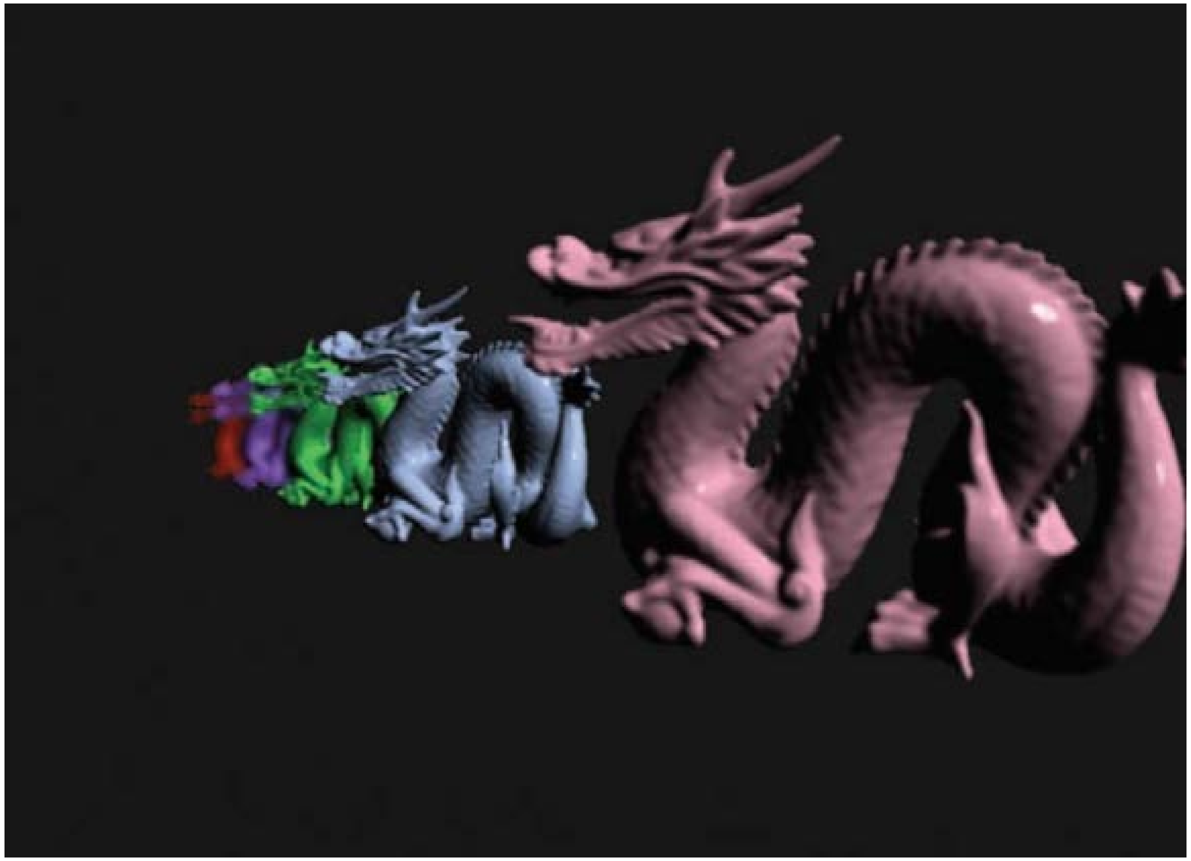


Other Usage

- Image Processing

Other Usage

- Depth of field



Other Usage

- AI flocking