ESCOLA DE NOVES TECNOLOGIES INTERACTIVES

centre adscrit a: UNIVERSITAT DE BARCELONA

Computer Graphics

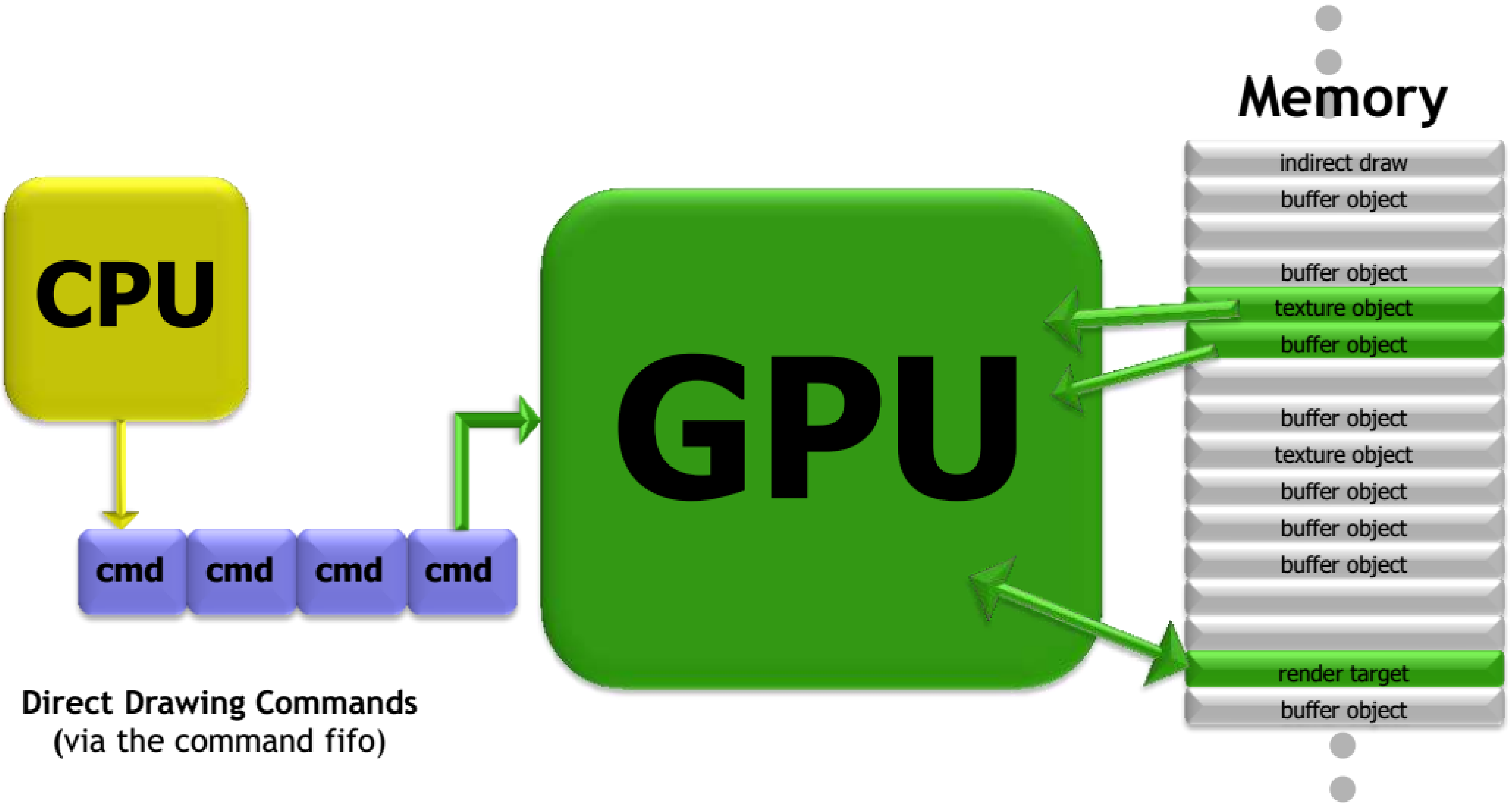
11. Modern OpenGL. Optimization and Compute Shaders

Dr Jesus Ojeda – jesusojeda@enti.cat

Contents

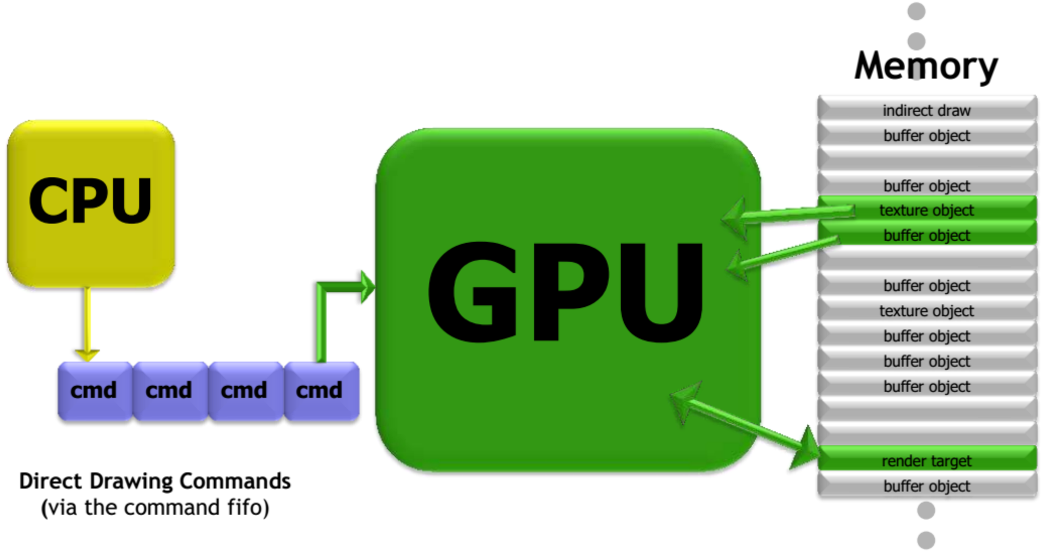
- Approaching Zero Driver Overhead
  - MultiDrawIndirect
  - Buffer Storage
  - Texture Arrays
  - Bindless and Sparse textures
- Compute Shaders

Why deprecate functionality?



Why deprecate functionality?

- Pros
  - Stable +20 year old code
  - Simple – driver handles troubles
- Cons
  - Demanding apps are restricted
  - Threading?
  - Hardware abstraction



This model is based on one CPU and commands are validated through the driver.

CPU-bound or call-intensive apps will suffer.

Intermission – Mantle, Vulkan & Direct3D 12

To reduce driver overhead:

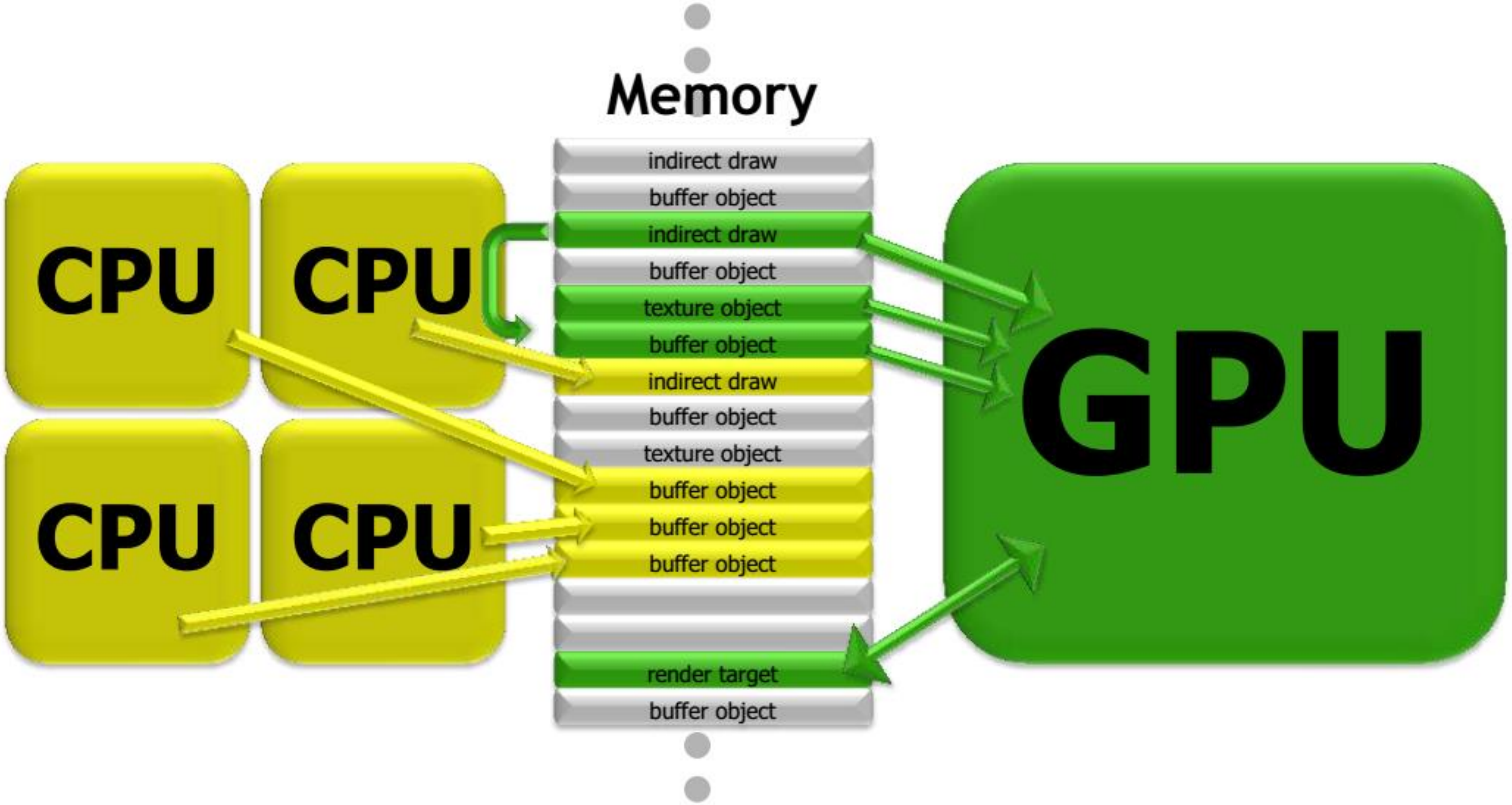
- Mantle. Developed by AMD and DICE, 2013.
- Vulkan. Developed by Khronos, based heavily on Mantle, 2014.
- Direct3D 12. Developed by Microsoft, 2014.
- Metal. Developed by Apple, 2014.

All are low-level based and provide full control to the developer.

Vulkan™

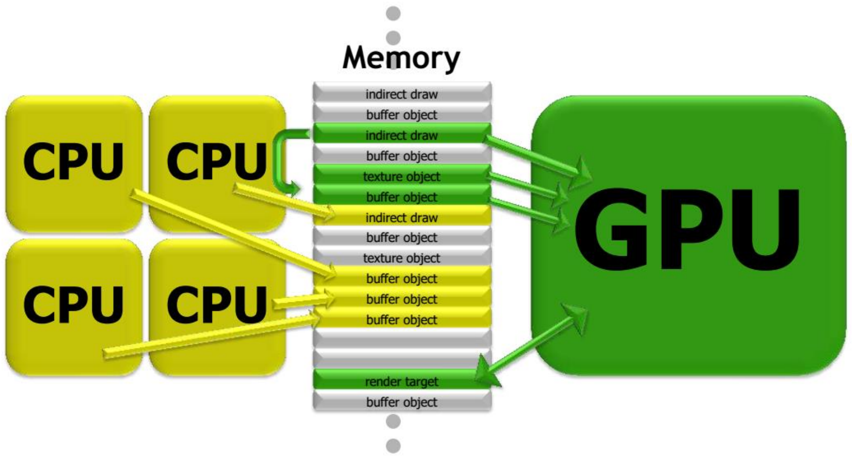
DirectX 12

Back to OpenGL. How to make it efficient?



OpenGL. How to make it efficient?

- CPU and GPU decoupled
- Multi-threaded
- GPU reads/writes commands from memory



No API, just using memory!

Approaching Zero Driver Overhead

- **MultiDrawIndirect**
- Buffer Storage
- Texture Arrays
- Bindless and Sparse textures

The Naïve Draw Loop

```
foreach( object )
{
    // bind framebuffer
    // set depth, blending, etc. states
    // bind shaders
    // bind textures
    // bind vertex/index buffers
    WriteUniformData( object );
    glDrawElements( GL_TRIANGLES, object->indexCount,
                    GL_UNSIGNED_SHORT, 0 );
}
```

Better Draw Loop

```
// sort or bucket visible objects
foreach( render target ) // framebuffer
foreach( pass ) // depth, blending, etc.
foreach( material ) // shaders
foreach( material instance ) // textures
foreach( vertex format ) // vertex buffers
foreach( object )
{
    WriteUniformData( object );
    glDrawElementsBaseVertex(GL_TRIANGLES,object->indexCount,
                             GL_UNSIGNED_SHORT,object->indexDataOffset,
                             object->baseVertex );
}
```

Drawing several of the same? Instancing

- glDrawArraysInstanced
- glDrawElementsInstanced

They require number of instances to be drawn as primcount parameter.

From the vertex shader you can access a counter (gl_InstanceID) that identifies in which instance you are.

Drawing several of the same? Instancing v2

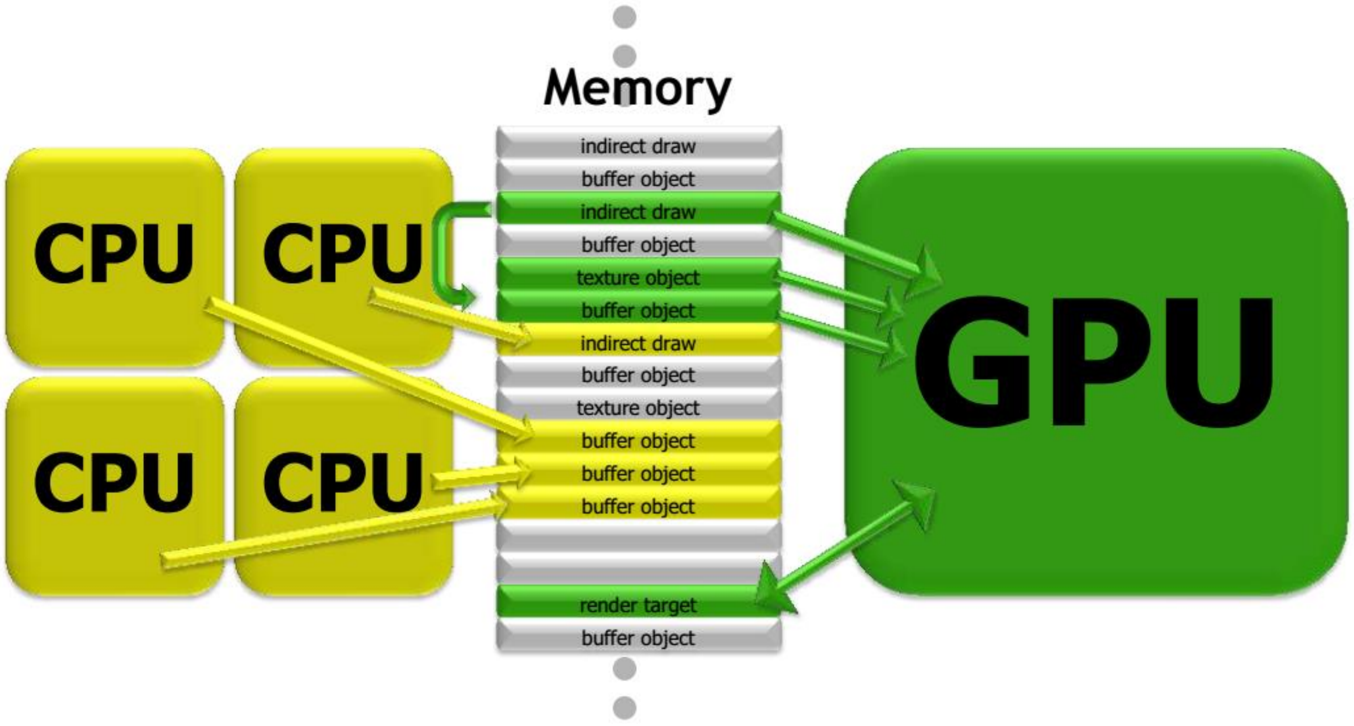
- glDrawArraysInstanced**BaseInstance**
- glDrawElementsInstanced**BaseInstance**

Same as before, **BUT** now each instance can source different attributes from memory (vertex buffers).

How these attributes change from instance to instance is controlled with glVertexAttribDivisor.

Even better. Whole scene in one call!

- We can use memory to define the multiple object draw call parameters.
- Then, in one call we can tell the GPU to use those parameters in memory to draw the multiple objects.
- MULTI DRAW INDIRECT



MultiDrawIndirect - Arrays

glMultiDrawArraysIndirect behaves similar to multiple calls to *glDrawArraysInstancedBaseIntance* but parameters are sourced from memory.

```
void glMultiDrawArraysIndirect(GLenum mode, const void *indirect, GLsizei drawcount, GLsizei
   stride);
typedef  struct {
   uint  count;
   uint  instanceCount;
   uint  first;
   uint  baseInstance;
} DrawArraysIndirectCommand;
```

An array in memory or in a GL_DRAW_INDIRECT_BUFFER buffer.

MultiDrawIndirect - Elements

glMultiDrawElementsIndirect behaves similar to multiple calls to *glDrawElementsInstancedBaseVertexBaseIntance* but parameters are sourced from memory.

```
void glMultiDrawElementsIndirect( GLenum mode, GLenum type, const void *indirect, GLsizei
    drawcount, GLsizei stride);
typedef  struct {
    uint  count;
    uint  instanceCount;
    uint  firstIndex;
    uint  baseVertex;
    uint  baseInstance;
} DrawElementsIndirectCommand;
```

An array in memory or in a GL_DRAW_INDIRECT_BUFFER buffer.

One MultiDraw to rule them all

```
DrawElementsIndirectCommand* commands = ...;
foreach( object )
{
    WriteUniformData( object, &uniformData[i] );
    WriteDrawCommand( object, &commands[i] );
}
glMultiDrawElementsIndirect(
                GL_TRIANGLES,
                GL_UNSIGNED_SHORT,
                commands,
                commandCount,
                0 );
```

Approaching Zero Driver Overhead

- MultiDrawIndirect
- **Buffer Storage**
- Texture Arrays
- Bindless and Sparse textures

Your typical buffer data loading and update

After glGenBuffers() and glBind():

void **glBufferData**(GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage);

and

void **glBufferSubData**(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid *data);

Better yet… MapBuffer

```
void* data = glMapBuffer(GL_ARRAY_BUFFER,
                         Offset,
                         dataSize,
                         GL_MAP_UNSYNCHRONIZED_BIT | GL_MAP_WRITE_BIT );

WriteGeometry( data, ... );

glUnmapBuffer(GL_ARRAY_BUFFER);
```

But doing these operations frequently causes overhead.
Remember we want to avoid as much call as we can.

Enter BufferStorage and Persistent Map

- Allocate buffer with glBufferStorage()

```
glBufferStorage(GL_ARRAY_BUFFER, size, NULL, flags);
```

- Use flags to enable persistent mapping

```
GLbitfield flags = GL_MAP_WRITE_BIT
                 | GL_MAP_PERSISTENT_BIT
                 | GL_MAP_COHERENT_BIT;
```

The buffer is kept mapped and writes from CPU are automatically visible to GPU.

Persistent Map v2

- Map once at creation time

data = **glMapBufferRange**(ARRAY_BUFFER, 0, size, flags);

- No more Map/Unmap in your draw loop (just write to data)
  - But need to do synchronization yourself
    - glMemoryBarrier() and glFenceSync()
    - glClientWaitSync()
    - glFinish()

Approaching Zero Driver Overhead

- MultiDrawIndirect
- Buffer Storage
- **Texture Arrays**
- Bindless and Sparse textures

How textures are used in OpenGL?

1. Create them

   void **glGenTextures**(GLsizei n, GLuint * textures);
2. Bind them

   void **glBindTexture**(GLenum target, GLuint texture);
3. Load data

   void **glTexImage2D**(GLenum target, GLint level, GLint internalFormat, GLsizei width,
   GLsizei height,GLint border, GLenum format, GLenum type, const GLvoid * data);
4. Assign to texture unit and use from shader

   void **glBindTextureUnit**(GLuint unit, GLuint texture);

   And from shader:

   uniform sampler2D sampler;

Better texture performance? Texture Arrays!

We can group textures with same shape (dimensions, format, mip-maps…) into texture arrays.

Then we will bind all textures at once.

2D textures into arrays will work as a 3D texture

```
glGenTextures(1,&texture);
glBindTexture(GL_TEXTURE_2D_ARRAY, texture);
// Allocate the storage.
glTexStorage3D(GL_TEXTURE_2D_ARRAY, mipLevelCount, GL_RGBA8, width, height, layerCount);
```

Then, from shader

```
uniform sampler2Darray textureArray;
//main...
color = vec4(texture(textureArray, vec3(TexCoords.xy, layer)));
```

Approaching Zero Driver Overhead

- MultiDrawIndirect
- Buffer Storage
- Texture Arrays
- **Bindless and Sparse textures**

Bindless textures. The why.

Textures are bound to numbered units in OpenGL. Binding has a cost, plus:

- There is a limited number of units (limited textures at once)
- State change between draw calls (expensive)
- Driver controls residency (which texture lives in GPU and which do not)

So, why not remove texture bindings?

Bindless textures. The how. (ARB)

```
// Create textures as normal, get handles from textures
GLuint64 handle = glGetTextureHandleARB(tex);
// Make resident
glMakeTextureHandleResidentARB(handle);
// Communicate 'handle' to shader... somehow
foreach (draw) {
    glDrawElements(...);
}
```

- No texture binds between draws!!
- In shader you use them as typical samplers. The only problem is how to get the handles in the shader.

Bindless textures. The how. (ARB) v2

How to get the handles to the shader?

Handles are 64-bit integers. Some solutions:

- Direct handle use

void **glUniformHandleui64ARB**(GLint location, GLuint64 value); ///FROM CODE

layout(**bindless_sampler**) uniform sampler2D bindless;   ///FROM SHADER

- Use uint64_t and cast to sampler (also from uniform buffers)

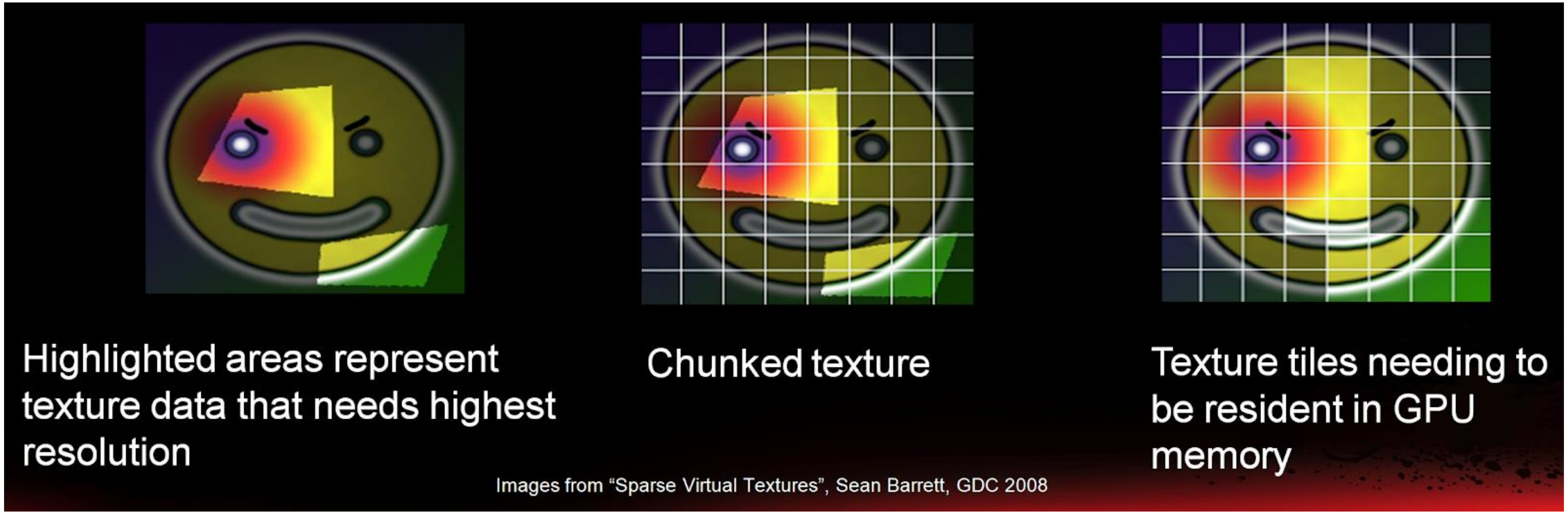
sampler2D sampl = sampler2D(some_uint64); ///FROM SHADER

Sparse textures. What?

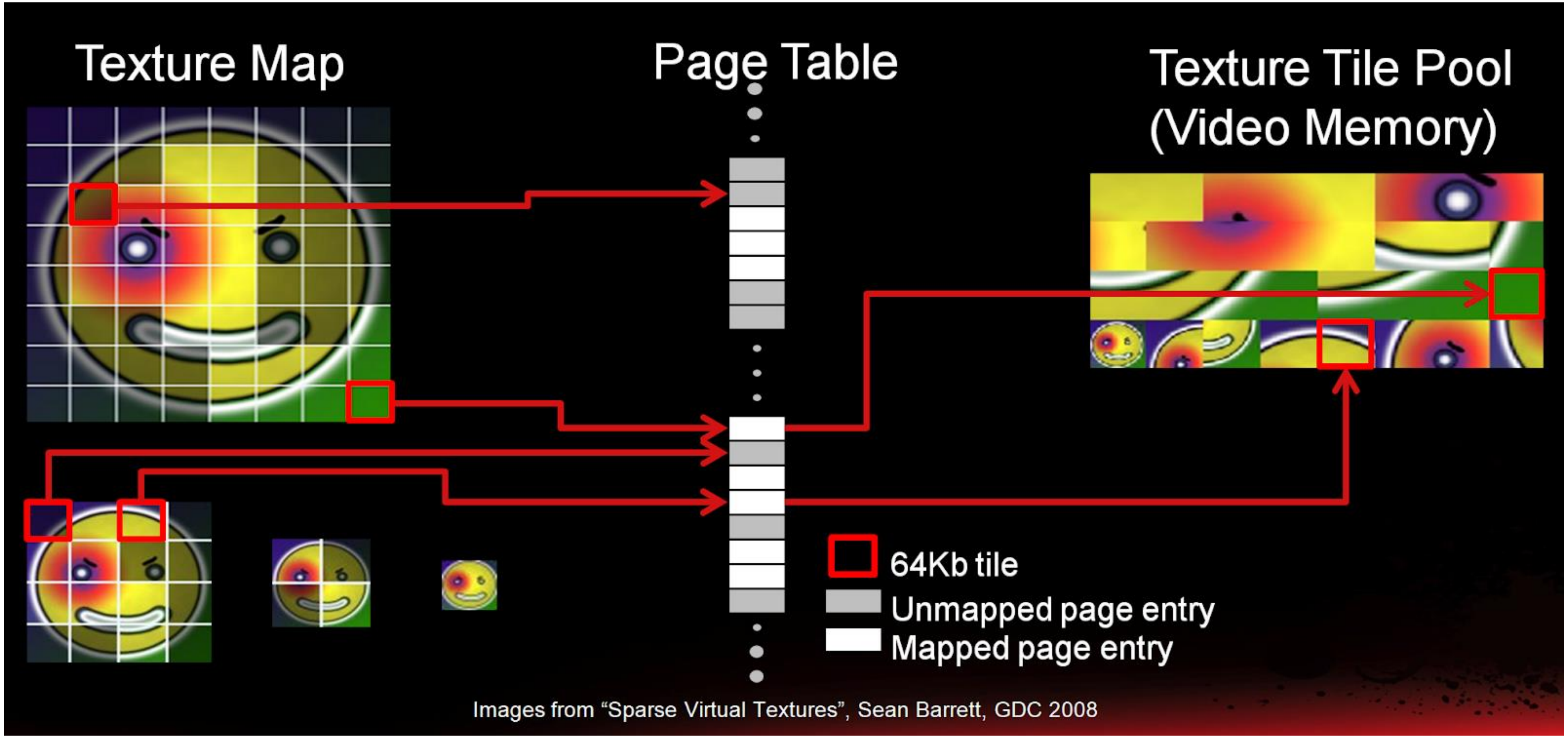
- What if we have textures larger than physical memory? We could not use them.
- We can separate virtual (much larger than available) from physical memory (limited) - operating system concept
- Then we stream data that we need on demand (not the whole texture would be visible at the most high level of detail).
- Also known as Virtual texturing or Partially Resident Textures.

Sparse textures (ARB)

Textures are arranged as tiles, which may be resident on GPU or not.



Sparse textures (ARB) v2



Sparse textures (ARB) v3

- So, you create a texture as usual and

```
// Tell OpenGL you want a sparse texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_SPARSE_ARB, GL_TRUE);
// Allocate storage
glTexStorage2D(GL_TEXTURE_2D, 10, GL_RGBA8, 1024, 1024);
```

- Now you have a virtual texture.

Sparse textures (ARB) v4

- Page sizes

```
// Query number of available page sizes
glGetInternalformativ(GL_TEXTURE_2D, GL_NUM_VIRTUAL_PAGE_SIZES_ARB, GL_RGBA8,
                      sizeof(GLint), &num_sizes);

// Get actual page sizes
glGetInternalformativ(GL_TEXTURE_2D, GL_VIRTUAL_PAGE_SIZE_X_ARB, GL_RGBA8,
                      sizeof(page_sizes_x),&page_sizes_x[0]);
glGetInternalformativ(GL_TEXTURE_2D, GL_VIRTUAL_PAGE_SIZE_Y_ARB, GL_RGBA8,
                      sizeof(page_sizes_y),&page_sizes_y[0]);

// Choose a page size
glTexParameteri(GL_TEXTURE_2D, GL_VIRTUAL_PAGE_SIZE_INDEX_ARB, n);
```

Sparse textures (ARB) v5

- Control page residency (commitment)

```
void glTexPageCommitmentARB(GLenum target, GLint level,
        GLint xoffset, GLint yoffset,
        GLint zoffset, GLsizei width,
        GLsizei height, GLsizei depth,
        GLboolean commit);
```

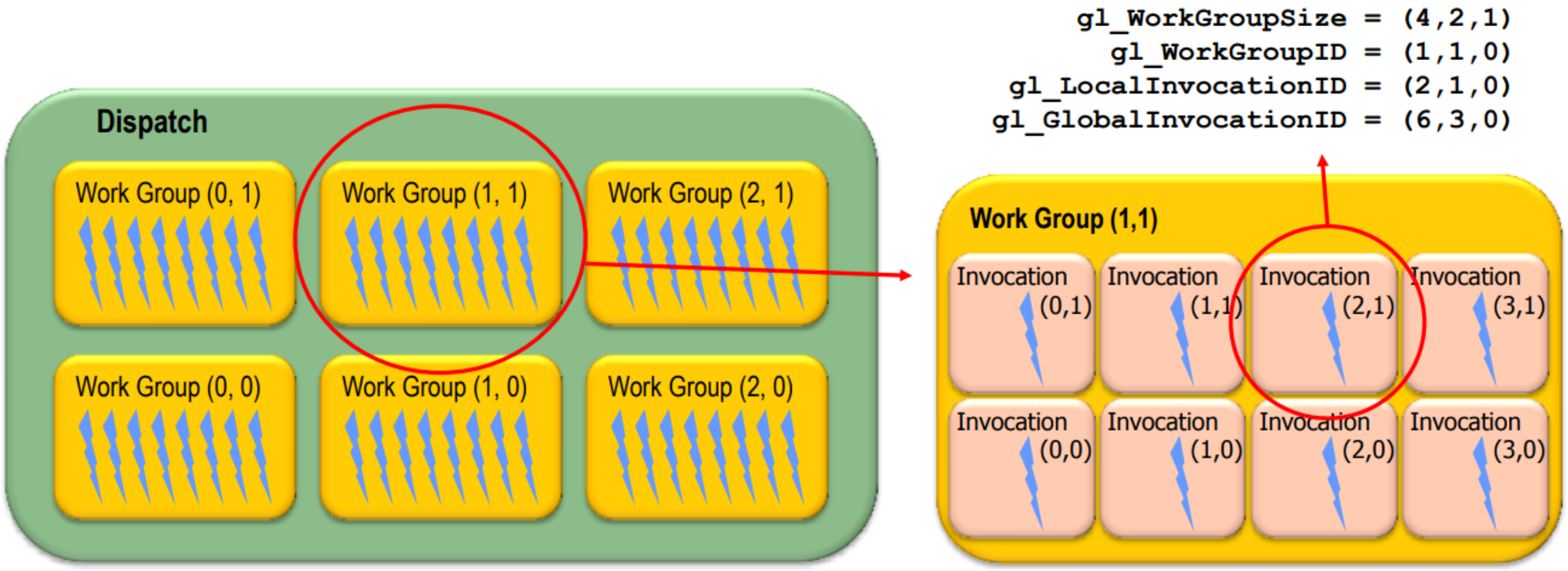
- Uncommited pages use no physical memory
- Commited pages may contain data

The rest of the usage of these kind of textures is as normal: data load, usage from shader, etc.

Compute Shaders

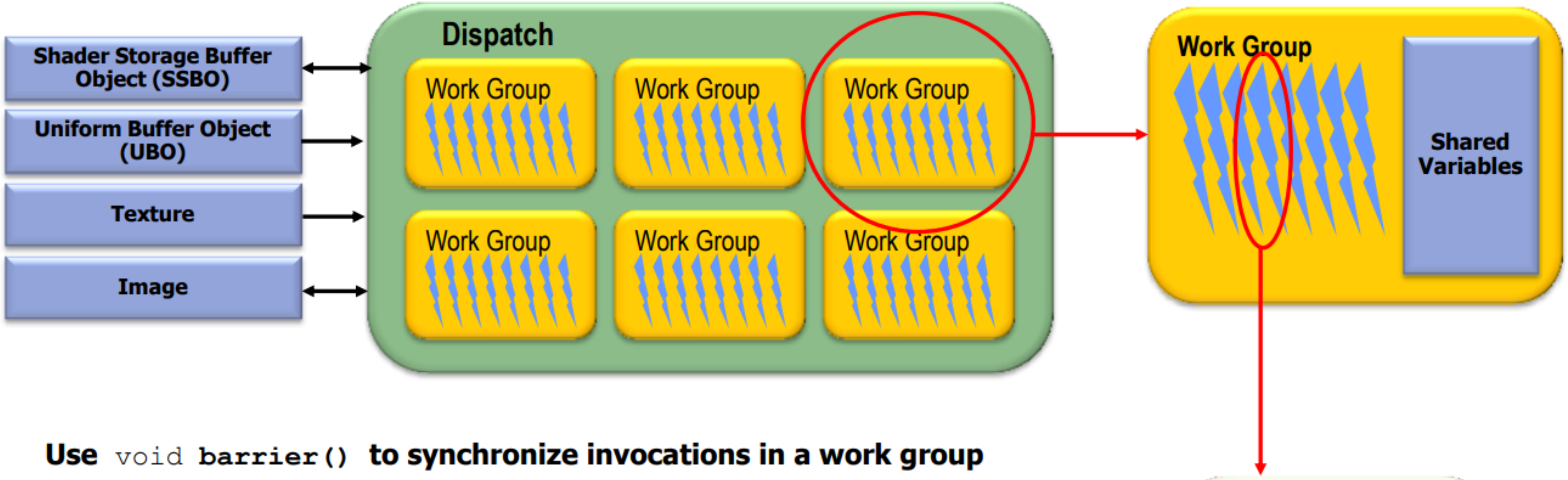
- What if we can use GPU for more than graphics?
- Physics? AI? Other computations?
- Similar to CUDA and OpenCL, which are frameworks for general computation on GPU, OpenGL now has compute shaders.
- Compute shaders are not part of the rendering pipeline and work in an abstract space.

Compute shaders. Execution model




```
   in uvec3 gl_NumWorkGroups;       // Number of workgroups dispatched
const uvec3 gl_WorkGroupSize;       // Size of each work group for current shader
   in uvec3 gl_WorkGroupID;         // Index of current work group being executed
   in uvec3 gl_LocalInvocationID;   // index of current invocation in a work group
   in uvec3 gl_GlobalInvocationID;  // Unique ID across all work groups and invocations
```

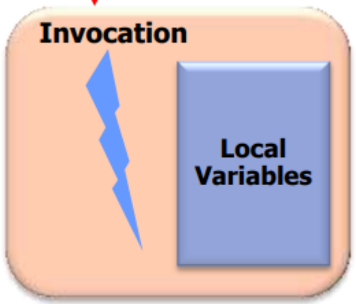
Compute shaders. Memory model



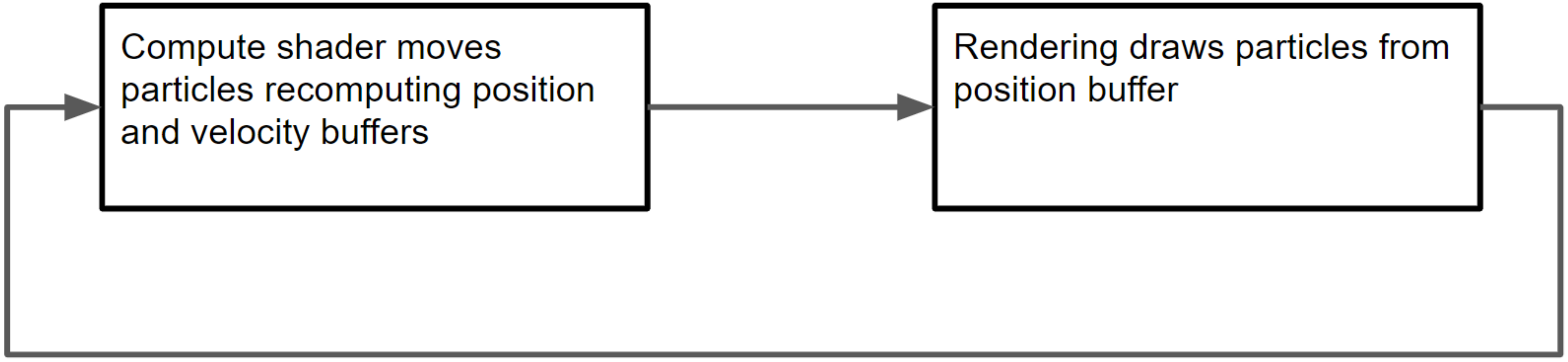
Use `void barrier()` to synchronize invocations in a work group

Use memory barriers to order reads/writes accessible to other invocations

```
void memoryBarrier();
void memoryBarrierAtomicCounter();
void memoryBarrierBuffer();
void memoryBarrierImage();
void memoryBarrierShared();        // Only for compute shaders
void groupMemoryBarrier();         // Only for compute shaders
```



Compute shaders. A particle system.



Compute shaders. A particle system. Buffers.

```
glGenBuffers(2, SSbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, SSbo[0]);
glBufferData(GL_SHADER_STORAGE_BUFFER, NUM_PARTICLES * sizeof(float) * 4, NULL, GL_STATIC_DRAW);
float *pos = reinterpret_cast<float*>(glMapBufferRange(GL_SHADER_STORAGE_BUFFER, 0, NUM_PARTICLES* sizeof(float) * 4,
                                                        GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT));
for (int i = 0; i < NUM_PARTICLES; ++i) {
        *pos++ = rnd_dist(rnd_gen)*10.f - 5.f,
        *pos++ = rnd_dist(rnd_gen)*5.f + 5.f,
        *pos++ = rnd_dist(rnd_gen)*10.f - 5.f;
        *pos++ = 1.f;
}
glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);

//Similarly for SSbo[1] -> Velocities
//…

//Also bind SSbo[0] to draw the particles
glGenVertexArrays(1, &Vao);
glBindVertexArray(Vao);
glBindBuffer(GL_ARRAY_BUFFER, SSbo[0]);
glVertexAttribPointer((GLuint)0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Compute shaders. A particle system. Invoking.

On Init create the program (just one compute shader per compute program!!)

```
c_shader = compileShader(compute_shader, GL_COMPUTE_SHADER);
c_program = glCreateProgram();
glAttachShader(c_program, c_shader);
linkProgram(c_program);
```

On the render loop, invoke it

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, SSbo[0]);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, SSbo[1]);
//////Compute
glUseProgram(c_program);
glUniform4fv(glGetUniformLocation(c_program, "SphPos"), 1, &SphPos.x);
glUniform1f(glGetUniformLocation(c_program, "DT"), dt);
glDispatchCompute(NUM_PARTICLES / WGROUP_SIZE, 1, 1);
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
//////Render
glUseProgram(Sphere::sphereProgram);
//Uniforms...
glBindVertexArray(Vao);
glDrawArrays(GL_POINTS, 0, NUM_PARTICLES);
glBindVertexArray(0);
glUseProgram(0);
```

Compute shaders. A particle system.

```
#version 430
layout( std430, binding=0 ) buffer Pos {vec4 pos[];};
layout( std430, binding=1 ) buffer Vel {vec4 vel[];};
layout( local_size_x = 128, local_size_y = 1, local_size_z = 1 ) in;
uniform vec4 SphPos;
uniform float DT;
const float eps = 0.4;
const float G = 9.81;

void main() {
uint gid = gl_GlobalInvocationID.x;
vec4 p = pos[gid];
vec4 v = vel[gid];
vec4 grav = G * normalize(vec4(SphPos.xyz, 1.) - p);
vec4 np = p + v * DT;
vec4 nv = v + grav * DT;
if(length(np.xyz - SphPos.xyz) < SphPos.w) {
vec4 onS = onSph(SphPos.xyz, SphPos.w, p.xyx, np.xyz); vec4 n = normalize(onS - SphPos);
float d = -dot(n, onS);
np = np - (1+eps)*(dot(np, n)+d)*n;
nv = nv - (1+eps)*(dot(nv, n))*n;
}
pos[gid] = np;
vel[gid] = nv;
}
```

```
vec4 onSph(vec3 sph, float rad, vec3 pre_x, vec3 pos_x) {
float cc = dot(sph, sph);
float pp = dot(pre_x, pre_x);
float cp = dot(sph, pre_x);
float pq = dot(pre_x, pos_x);
float cq = dot(sph, pos_x);
float qq = dot(pos_x, pos_x);
float a = qq + pp - 2 * pq;
float b = 2 * cp + 2 * pq - 2 * cq - 2 * pp;
float c = cc + pp - 2 * cp - rad * rad;
float alpha_p = (-b + sqrt(b*b - 4 * a*c)) / (2 * a);
float alpha_m = (-b - sqrt(b*b - 4 * a*c)) / (2 * a);
float alpha = alpha_p;
if (0.0 < alpha_m && alpha_m < 1.0)
alpha = alpha_m;
return vec4(
pre_x + ((pos_x + (-pre_x)) * alpha), 1.0);
}
```

Resources

- Graham Sellers, Richard S. Writght, Jr. Nicholas Haemel. **OpenGL SuperBible**, 6th Edition. Pearson education.
- John Kessenich, Graham Sellers, Dave Shreiner. **OpenGL Programming guide**. Ninth Edition. Pearson Education.
- https://www.youtube.com/watch?v=PPWysKFHq9c
- https://www.youtube.com/watch?v=K70QbvzB6II
- https://archive.org/details/GDC2013McDonald
- https://developer.nvidia.com/opengl-vulkan
- https://www.khronos.org/registry/OpenGL-Refpages/gl4/
- https://www.khronos.org/opengl/wiki/Main_Page