Lesson 7 – Particle systems
Compute shaders, Geometry shaders

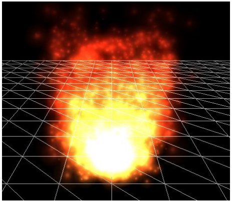
PV227 – GPU Rendering

Jiří Chmelík, Jan Čejka
Fakulta informatiky Masarykovy univerzity

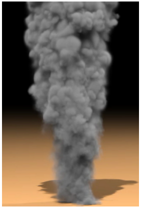
24. 10. 2018

Particle systems

Particle systems are used for many effects:



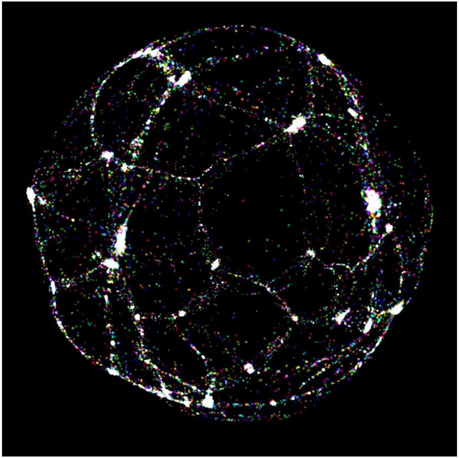
Fire



Smoke

water, wind, explosions, debris, leaves, birds, ...

N-body simulation



N-Body simulation

Physics behind

- Force between particles:

$$F = G\frac{m_1 m_2}{r^2}$$

- Acceleration:

$$a = \frac{F}{m}$$

- Position:

$$x = \int a \, \mathrm{d}t^2$$

Physics behind

- Force from particle p_{other} to partice p:

$$|F| = \frac{constant}{\|p_{other} - p\|^2}$$
$$\text{direction of } \vec{F} = \text{direction of } (p_{other} - p)$$

- Acceleration:

$$a = constant \cdot \sum F$$

- Position:

$$x_1 = x_0 + v_0 \Delta t + \frac{1}{2} a \Delta t^2$$
$$v_1 = v_0 + a \Delta t$$

Physics – pseudocode

```
foreach particle p do
    x_0 ← read p's position
    v_0 ← read p's velocity
    accel ← (0, 0, 0)
    foreach other particle other do
        x_other ← read other's position
        direction ← x_other − x_0
        dist^2 ← dot(direction, direction)
        if dist^2 > threshold then
            accel ← accel + normalize(direction)/dist^2
        end
    end
    accel ← accel · accel_factor
    x_1 ← x_0 + v_0 Δt + (1/2) accel Δt^2
    v_1 ← v_0 + accel Δt
    store x_1
    store v_1
end
```

Task: Implement N-body simulation

- **Task 1:** Implement N-body simulation on CPU
  - See the comments in C++ code for the names of variable and constants
  - Don't forget there are two arrays with particle positions, one to read from and one to write into
  - The complexity is $\mathcal{O}(n^2)$, test on low number of particles. Once it all works, try *Release* build.

General Purpose GPU (GPGPU)

- Motivation: Use those many threads on GPU to speed up our computation.
- In this lecture, we will describe the very basics of GPGPU. For more information:
  - Loop up CUDA or OpenCL on the Internet
  - See PV197 GPU Programming

History of GPGPU

- Brief history:
  - Since cca 2000: fragment shaders
  - Since cca 2006: CUDA, OpenCL
  - Now: Compute shaders

Basic principles of compute shaders

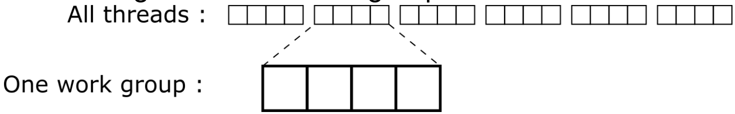
- Similar to vertex/fragment shaders:
  - Many (mostly independent) threads
  - Threads do (mostly) the same
- Different from vertex/fragment shaders:
  - VS/FS processes one vertex/fragment
  - Compute shaders may process whatever
  - Each thread may process any number of items
  - Threads can share the mid-results of the computation
- Reading and writing data
  - Buffers via SSBO
  - Textures via image load/store
  - Atomic operations
  - OK, available in other shaders too
- Can do (mostly) whatever, so **beware of bugs in the code**

Support in OpenGL

- GLSL code like in other shaders:
  - Access to uniform variables, UBOs, SSBOs, textures
  - Structures *vec4*, *mat4*, ...
  - Functions *dot*, *cross*, ...
  - Runs the code in *main* function
- Loading and using similarly as other shaders
  - *glCreateShader(GL_COMPUTE_SHADER)*
  - Attaching to programs, using programs
- Outside rendering pipeline
  - Use *glDispatchCompute* instead of *glDraw**

Organization of threads

- Threads are organized into work groups:



6 work groups, 24 threads

- Threads in work group can share data via shared memory
- Threads can be organized in 1D, 2D, and 3D. We will use 1D.
- Up to 1024 threads in one work group.
- Up to 65536 work groups.

Indexing of threads

- Specifying number of threads in work group:
  In GLSL: *layout (local_size_x = 256) in;*
- Specifying number of work groups:
  In C++: *glDispatchCompute(#_of_work_groups_in_x, 1, 1);*
- Index of a thread in its work group:
  In GLSL: *gl_LocalInvocationID.x*
- Index of a thread in all work groups:
  In GLSL: *gl_GlobalInvocationID.x*
- Index of the work group a thread is a part of
  In GLSL: *gl_WorkGroupID.x*
- Size of one work group (as specified with layout):
  In GLSL: *gl_WorkGroupSize.x*
- Number of work groups (as specified with glDispatchCompute):
  In GLSL: *gl_NumWorkGroups.x*

```
gl_GlobalInvocationID.x :   0  1  2  3   4  5  6  7   8  9 10 11
gl_LocalInvocationID.x  :   0  1  2  3   0  1  2  3   0  1  2  3
gl_WorkGroupID.x        :   0  0  0  0   1  1  1  1   2  2  2  2
layout (local_size_x = 4) in;
glDispatchCompute(3, 1, 1);
gl_WorkGroupSize = uvec3(4, 1, 1)
gl_NumWorkGroups = uvec3(3, 1, 1)
```

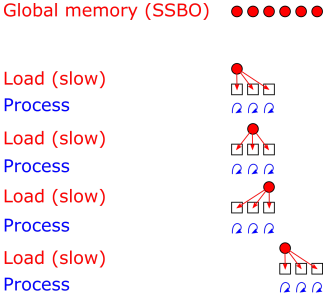
Task: Rewrite to compute shaders

- **Task 2:** Implement N-body simulation in compute shaders
  - See the comments in the code for the names of variable and constants
  - Use one thread to compute one particle.
  - Copy and paste the code from C++ and do minor changes

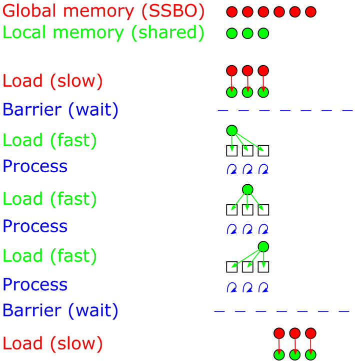
Sharing data between threads

- Sharing via shared memory, can be shared only between threads in the same work group.
- Specification in GLSL:
  shared variable_type variable_name;
- Stored values are visible to other threads
- Threads run in parallel (!), so we must synchronize the threads
- GLSL function *barrier()*
  - Calling thread waits until all other threads in the work group reach the barrier
  - After the barrier, all threads can read the new values in shared variables
  - After the barrier, no threads will need the old data in shared variables

Sharing data between threads – diagram



Without shared memory



With shared memory

Sharing data between threads – pseudocode

```
foreach particle p do
    ...
    foreach gl_WorkGroupSize.x of other particles do
        read position of one particle into shared memory
        barrier() – wait until all other threads read their positions
        foreach other particle other in shared memory do
            process the particle
        end
        barrier() – wait until all other threads finish processing the data
    end
    ...
end
```

Task: Share data between threads

- **Task 3:** Share the positions between threads in work group
  - Copy the code from *nbody_compute.glsl* to *nbody_shared_compute.glsl* and rewrite it
  - See the comments in the code for the names of variable and constants

Pros and cons of using compute shaders

- When compared to CPU:
  - Pros: many threads, the data stays on GPU
  - Cons: threads must run mostly the same code
- When compared to other shaders
  - Pros: more flexible
  - Cons: more difficult
- When compared to CUDA / OpenCL
  - Pros: native access to buffers / textures
  - Cons: less flexible

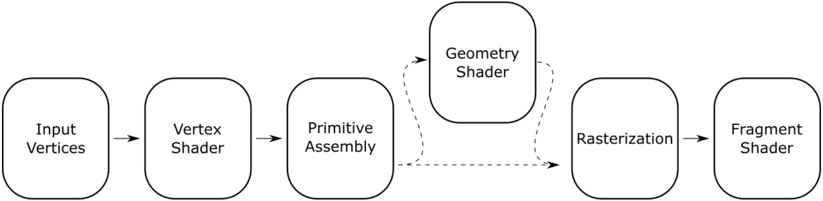
glMemoryBarrier

glMemoryBarrier

- When the data is updated using outputs from vertex/fragment shaders, memory copies etc., OpenGL knows which data is update, what operations must wait and what operations may be executed in parallel.
- When we load/store the data using SSBO or texture images (in compute or other shaders), OpenGL does not know what was done. Delaying all operations may not be necessary.
- Use *glMemoryBarrier* to tell OpenGL which memory reads depend on the result of the (not only compute) shaders.
- Look up its usage in *Cv7_main.cpp*.

Geometry shaders

- New programmable stage (optional)
- Between vertex shader and fragment shader
- Takes the whole primitive on input
- Creates new primitives on output
- Use *GL_GEOMETRY_SHADER* in C++ to create a geometry shader



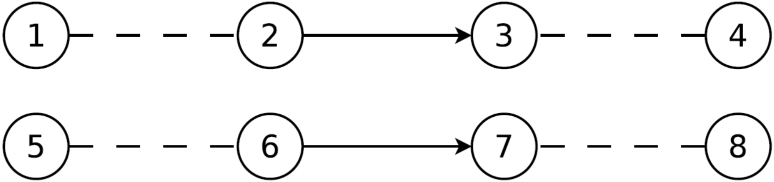
Input Primitives

- Defined in GLSL code:
  layout (primitive_type) in;
- Five supported types, each corresponds with different number of vertices visible on input

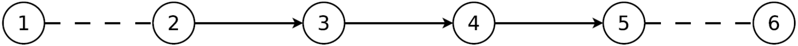
primitive	#vertices
points	1
lines	2
lines_adjacency	4
triangles	3
triangles_adjacency	6

- Primitive type must match the draw command
  - Input triangles, drawing triangles: OK
  - Input triangles, drawing triangle strip: OK
  - Input points, drawing triangles: not OK

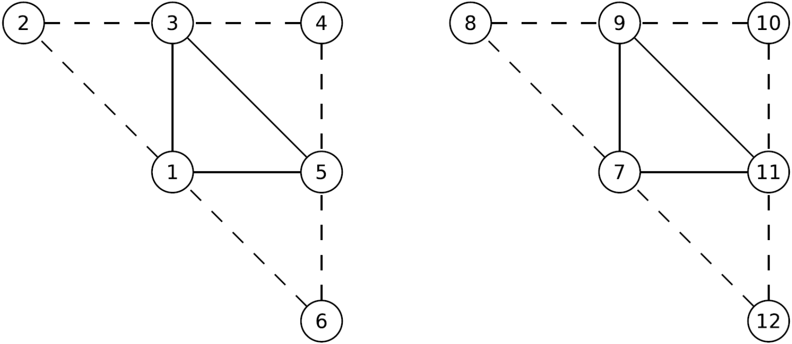
Additional OpenGL primitives



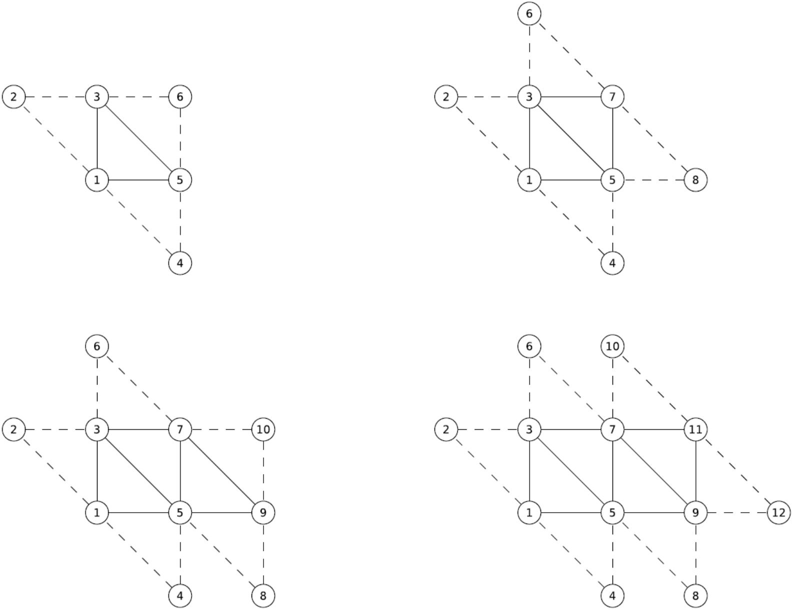
GL_LINES_ADJACENCY



GL_LINE_STRIP_ADJACENCY



GL_TRIANGLES_ADJACENCY



GL_TRIANGLE_STRIP_ADJACENCY

Output Primitives

- Three options: *points*, *line_strip*, *triangle_strip*
- Geometry shader must also specify maximum number of vertices that can be generated.
- Specification in GLSL:
  layout (triangle_strip, max_vertices = 4) out;
- Input primitive needs not to correspond with output primitive
- Input primitive is discarded

Input Data

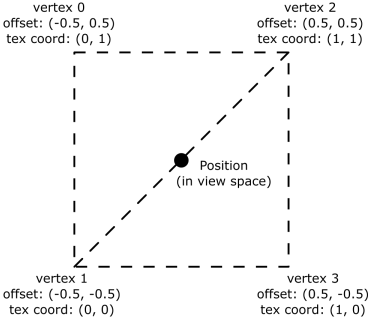
- Data from vertex shader, in arrays.
- Size of the array corresponds to the number of vertices of the input primitive.
- Build-in variables in array *gl_in*, e.g.:
  gl_in[0].gl_Position
- Other variables must be defined as arrays, e.g.:
  in VertexData { ... } inData[];
- Size of the array may either be not specified, or must correspond to the number of vertices of the primitive.

Output Data

- Output data specified in the same way as in vertex shader.
- Once all data of a vertex is specified, call *EmitVertex()*
- Always define values of all output variables!
- Primitive can be closed and restarted with *EndPrimitive()*

Example: Render points as textured quads

- Use geometry shaders to render quads with texture in place of points.
- Input primitive is point
- Output primitive is one triangle strip of four vertices
- Positions and texture coordinates can be computed very well in view space:



Task: Render points as textured quads

- **Task 4:** Use geometry shaders to render points as quads
  - In vertex shader, transform the position into view space, and pass the color.
  - In geometry shader, derive the position, texture coordinate and color of each vertex, and compute *gl_Position*
  - Fragment shader is done.

More on geometry shaders

In the next lecture . . .