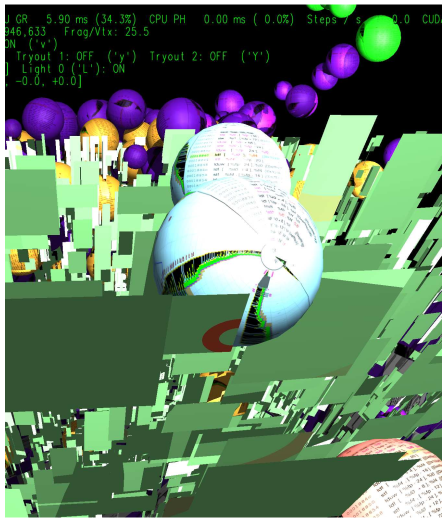
LSU EE 4702-1 Homework 5 Solution Due: 27 November 2017

The solution code is in the repository in file `hw05-shdr-sol.cc`. Running the makefile will generate an executable for the solution named `hw05-sol`. (Strictly speaking, it's a second name for `hw05`. When run it checks its own name and if it starts `hw05-sol` it uses the solution shader, otherwise it uses the pre-solution shader.)

Colorized solution files can be found at `http://www.ece.lsu.edu/koppel/gpup/2017/hw05-shdr-sol.cc.html`. The CPU code (which is not modified for the solution) can be found at `http://www.ece.lsu.edu/koppel/gpup/2017/hw05.cc.html`.

Problem 0: Follow the instruction on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should initially show Scene 1, a cloud of green rectangles, *tiles*, with balls dropping on them. (The familiar water wheel is in scene 2.)



```
U GR   5.90 ms (34.3%)  CPU PH   0.00 ms ( 0.0%)  Steps / s    0.0  CUD
946,633   Frag/Vtx: 25.5
ON  ('v')
  Tryout 1: OFF  ('y')  Tryout 2: OFF  ('Y')
]  Light 0 ('L'): ON
, -0.0, +0.0]
```

Scene Interaction and User Interface

Pressing `d` (drip) will toggle ball dripping on and off. Pressing `x` will toggle a shower of balls. Pressing digits `1` through `2` will initialize different scenes, the program starts with scene 1. Pressing `t` starts the lots of balls scene (t is for test). Pressing `T` drops fewer balls.

The scenes differ in the number of objects, which include spheres, tiles, and the platform (which for this assignment we'll consider one object). The rendering of objects by type can be toggled on and off by pressing `!`, `@`, `#`, for spheres, tiles, and the platform. See the green text line starting with Hide. The scene includes shadow and mirror effects.
These can be toggled by pressing `o` and `r`, their state is shown next to Effect.

Display of Performance-Related Data

The top green text line shows performance. XF shows the number of display updates per frame buffer update. An ideal number is 1. A 2 means that two display updates were done in the time needed to update the frame buffer, presumably because the code could not update the frame buffer fast enough. GPU.GL shows how long the GPU spends updating the frame buffer (per frame), GPU.CU shows how long the computational accelerator takes per frame. The computational accelerator computes physics in this assignment. On the lab computers the computational accelerator GPU is different than the one performing graphics. CPU GR is the amount of time that the CPU spends on graphics, and CPU PH is the amount of time that the CPU spends on physics.

The second line, the one starting with Vertices, shows the number of items being sent down the rendering pipeline per frame. Clip Prim shows the number of primitives before clipping (in) and after clipping (out).

User Interface for Navigation and Interaction

Initially the arrow keys, PageUp, and PageDown can be used to move around the scene. Press (lower-case) `b` and then use the arrow and page keys to move the first ball around. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

Pressing `Ctrl+` increases the size of the green text and `Ctrl+` decreases it. The `+` and `-` keys can be used to change the value of certain variables to change things like the light intensity, spring constant, and variables needed for this assignment. The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text. Pressing `Tab` cycles through the different variables. Those who want to increase the spring constant to the point that the scene explodes may be disappointed to learn that there is a protection mechanism that increases the mass of balls when the spring constant is high enough to make the system go unstable, such balls turn red.

Look at the comments in the file `hw05.cc` and `hw05-shdr.cc` for documentation on other keys.

Switching Between Shaders

The program can use three different sets of shaders for the spheres, PLAIN (tiles_0), PROBLEM 1 (tiles_1), and PROBLEM 2 (tiles_2). Pressing `v` switches between them.

Description of Rendering

Scene 1 is filled with 8000 light green tiles arranged randomly in some volume, but all facing one point. The CPU code rendering tiles (all tiles, not just the light green ones described above) is in routine `World::render_tiles` in file `hw05.cc`. This routine launches one of the three shaders for rendering tiles. Shader `s_hw05_tiles_0`, which will be called `tiles_0` here, is used with a rendering pass in which the vertices are grouped as individual triangles. The shaders for this version are in file `hw05-shdr.cc` and have names ending in `tiles_0`. The `tiles_0` work correctly in the unmodified code, and they provide one special effect: a tile changes to red when touced by a ball and remains red for two seconds. The `tiles_0` shaders are for at most experimentation, they don't need to be modified for the homework. The `tiles_1` shaders are used in a rendering pass that also groups vertices as individual triangles, but the values used for `glVertex` and `glColor` are unconventional (see the code). The `tiles_2` shaders are used in a rendering pass in which vertices are not grouped (meaning points). For `tiles_2` the vertex data is placed in buffer objects, and the buffer objects are only updated when the underlying data changes. The only vertex shader input for the `tiles_2` is `gl_VertexID`. Initially the `tiles_1` and `tiles_2` shaders just call their `tiles_0` counterparts and so they won't render properly.

Debugging Help

Boolean variables `opt_tryout1` and `opt_tryout2` can be used for debugging on the CPU. Pressing `y` and `Y` toggles their state. In the shader code use `tryout.x` and `tryout.y` for these two variables.

To see the difference between your code and the original code issue the Emacs command `Ctrl x v =` or from the shell the command `git diff`.

Problem 1: Modify the `tiles_1` shaders in `hw05-shdr.cc` so that they correctly render the tiles. They must operate with the data provided by `World::render_tiles` case 1, unconventional though it is. Do not change what data is provided by the CPU for this rendering pass.

- Work with the data provided from the CPU, don't send additional information.
- The code must be reasonably efficient.
- Pay attention to the geometry shader layout declarations **including** `max_vertices`.
- Pay attention to the `w` component of vertex shader inputs. *Note: This warning was not in the original assignment.*

First, consider the CPU code for setting up the rendering pass, which is case 1 in `World::render_tiles(bool simple)`:

```
  case 1:
    {
      pShader_Use use(s_hw05_tiles_1);
      glUniform2i(1, opt_tryout1, opt_tryout2);
      glUniform1i(2, light_state_get());
      glUniform1f(3, world_time);

      glBegin(GL_TRIANGLES);
      for ( Tile* tile: tiles )
        {
          glColor3fv(tile->color);
          glVertex3fv(tile->pt_00);
          glColor4fv(tile->tact_pos);
          glVertex3fv(tile->ax);
          glVertex3fv(tile->ay);
        }
      glEnd();
    }
    break;
```

Inspection of the CPU code for case 1 reveals that:

- The primitive used in the rendering pass is individual triangles.
- Vertex 0 of each triangle is given the color in the `gl_Color` vertex shader input and the coordinate of `pt_00` in the `gl_Vertex` input.
- Vertex 1 of each triangle is given the contact position and time in the `gl_Color` input and the tile's `ax` **vector** in the `gl_Vertex` input
- Vertex 2 of each triangle is given the tile's `ay` **vector** in the `gl_Vertex` input
- Unlike the case 0 code, the **tile normal is not provided**.

From this we need to plan the design of the shaders.

Vertex Shader: Typically the vertex shader performs operations that can be done on individual vertices, such as transforming the vertex coordinate from object to clip space. But that won't work here because the kind of data in the shader input is different for each triangle vertex. For example, a transformation from object to clip space using the modelview matrix would work for the coordinate in vertex 0 but not for the vectors in vertices 1 and 2 of each triangle. Therefore all the vertex shader can do is pass the data to the geometry shader. The vertex shader code appears below:

```
vs_main_tiles_1() {
  // Pass through inputs unchanged.
  vertex_e = gl_Vertex;  // Either pt_00, ax, or ay.
  color = gl_Color;      // Either color or tact_pos_time.
}
```

Geometry Shader: The geometry shader needs to compute the coordinates of the tile corners (vertices) and also compute the normal.

Because the vectors `ax` and `ay` were sent using `glVertex3f` their `w` component was automatically set to 1. That's correct for coordinates but wrong for vectors, for which the `w` component, if present at all, should be 0. To see why consider adding coordinate $\begin{bmatrix}1\\2\\3\end{bmatrix}$ to vector $\begin{bmatrix}0\\40\\0\end{bmatrix}$. The sum is $\begin{bmatrix}1\\2\\3\end{bmatrix}+\begin{bmatrix}0\\40\\0\end{bmatrix}=\begin{bmatrix}1\\42\\3\end{bmatrix}$. This works if the coordinate is a homogenized homogeneous coordinate $\begin{bmatrix}1\\2\\3\\1\end{bmatrix} + \begin{bmatrix}0\\40\\0\end{bmatrix} = \begin{bmatrix}1\\42\\3\\1\end{bmatrix}$. If the vector is represented by a four-component data type, such as **vec4**, the fourth, **w**, component must be set to zero. If it were set to 1 then the **sum would be wrong:** $\begin{bmatrix}1\\2\\3\\1\end{bmatrix} + \begin{bmatrix}0\\40\\0\\1\end{bmatrix} = \begin{bmatrix}1\\42\\3\\2\end{bmatrix} = \begin{bmatrix}0.5\\21\\1.5\\1\end{bmatrix}$.

In the geometry shader below the **w** components of the vectors are set to 0. Then the coordinates of the tile corners are computed and placed in an array. The tile normal is computed using the **ax** and **ay** vectors, converted to eye space and assigned to the shader output **normal_e**. The color from triangle vertex 0, which is actually a color, is assigned to shader output **color**. Finally a loop transforms and emits the four vertex coordinates.

```
void
gs_main_tiles_1()
{
  /// SOLUTION -- Problem 1

  vec4 vtx_o[4];

  // Write tile information into appropriately named variables.
  vec4 pt_00 = In[0].vertex_e;
  vec4 ax_o = vec4(In[1].vertex_e.xyz,0);
  vec4 ay_o = vec4(In[2].vertex_e.xyz,0);

  // Compute the object-space coordinates of the tile corners.
  vtx_o[0] = pt_00;
  vtx_o[1] = pt_00 + ax_o;
  vtx_o[2] = pt_00 + ay_o;
  vtx_o[3] = pt_00 + ay_o + ax_o;
  // The order is chosen for a triangle strip.

  // Compute the tile normal.
  vec3 normal_o = normalize(cross(ax_o.xyz,ay_o.xyz));
  normal_e = normalize(gl_NormalMatrix * normal_o);

  color = In[0].color;

  for ( int i=0; i<4; i++ ) {
      gl_Position = gl_ModelViewProjectionMatrix * vtx_o[i];
      vertex_e = gl_ModelViewMatrix * vtx_o[i];
      EmitVertex();
    }

  EndPrimitive();
}
```

Problem 2: Modify the code in `tiles_2` so that it renders the tiles correctly. They must operate with the data provided by `World::render_tiles` case 2, which is much more straightforward than case 1. Note that case 2 uses points as the primitive.

- Work with the data provided from the CPU, don't send additional information.
- The code must be reasonably efficient.
- Pay attention to the geometry shader layout declarations **including** `max_vertices`.

Inspection of the CPU code for case 2 and the shader code reveals the following:

- All of the data needed to render a tile have been placed in buffer objects which have been bound to arrays in the shader code. For example, the `pt_00` coordinates are bound to array `pt_00s`.
- A rendering pass is performed in which the primitives are points.
- The case 2 code does not provide any vertex shader inputs. Not even `gl_Vertex`.
- The rendering pass is initiated using `glDrawArrays(GL_POINTS,0,tiles.size());`, and so the vertex shader will be invoked with `gl_VertexID` ranging from 0 to `tiles.size()-1`.

As in the solution to Problem 1, all of the work will be done in the geometry shader. All the vertex shader will do is pass the vertex id to the geometry shader. Unlike the situation in Problem 1, we could in principle compute and transform tile coordinates in the vertex shader and then send them to the geometry shader. However, that would unnecessarily increase the amount of data sent between the vertex and geometry shaders, which would waste resources and perhaps slow down execution.
Vertex Shader: The vertex shader send the vertex id to the geometry shader using new vertex shader output `vertex_id`. The interface block and vertex shader appear below.

```
in Data_to_GS {
  // Note: Unneeded variables removed.
  int vertex_id;
} In[1];

void vs_main_tiles_2() {
  vertex_id = gl_VertexID;
}
```

Geometry Shader: The geometry shader reads tile data from the arrays rather than from shader inputs. The geometry shader also includes code for the solution to Problem 3. Otherwise it is the same as the geometry shader in the Problem 1 solution.

```
void gs_main_tiles_2() {
  int vertex_id = In[0].vertex_id;

  // Retrieve information about the tile from buffer objects instead of shader inputs.
  vec4 vtx_o[4];
  vec4 pt_00 = pt_00s[vertex_id];
  vec4 ax_o = vec4(axs[vertex_id].xyz,0);
  vec4 ay_o = vec4(ays[vertex_id].xyz,0);

  // Compute tile object space vertex coordinates.
  vtx_o[0] = pt_00;
  vtx_o[1] = pt_00 + ax_o;
  vtx_o[2] = pt_00 + ay_o;
  vtx_o[3] = pt_00 + ay_o + ax_o;

  // Compute normal.
  vec3 normal_o = normalize(cross(ax_o.xyz,ay_o.xyz));
  normal_e = normalize(gl_NormalMatrix * normal_o);

  // Convert contact position to eye space and write it to a
  // geometry shader output. Also write time. (Part of Problem 3 solution.)
  //
  vec4 tact_pos_time = tact_pos_times[vertex_id];
  tact_pos_time_e = gl_ModelViewMatrix * vec4(tact_pos_time.xyz,1);
  tact_pos_time_e.w = tact_pos_time.w;

  color = colors[vertex_id];

  for ( int i=0; i<4; i++ ) {
      gl_Position = gl_ModelViewProjectionMatrix * vtx_o[i];
      vertex_e = gl_ModelViewMatrix * vtx_o[i];
      EmitVertex();
    }
  EndPrimitive();
}
```

Problem 3: Modify the code in any of the shaders so that when a tile is struck by a ball an expanding red ring is drawn centered at the point of contact. The ring should be rendered using existing primitives, don't emit new primitives for the ring. The screenshot at the top of the assignment shows one ring in the center of the image, and some larger faded regions below.

Each tile has a `tact_pos` member that holds the coordinates of the point on the tile that the ball has struck, that in the `x`, `y`, and `z` components, and the time that the contact occurred, in the `w` component. All three rendering passes provide this data in one way or another, see `World::render_tiles`. Also, notice that `world_time` is provided to shaders.

The ring should be easily visible for a few seconds, until it expands to the point where the tile is completely inside (and so the ring is no longer visible).

- Don't omit new primitives, instead use the fragment shader operating on existing primitives for rendering the tile.
- The width of the ring should be some fixed object-space size, don't make it one pixel wide.
- Try to reduce the amount of work done by the fragment shader by doing extra work in the vertex or geometry shaders.
- Also try reducing the amount of work done by the rasterization stage.

To solve this the contact position coordinate and time will be sent to the fragment shader. The fragment shader can then compute the distance between the fragment coordinate (which will be in `vertex_e`) and the contact coordinate. The problem statement did not give a particular ring expansion speed or width. The solution uses a speed of 0.5 object-space units per second and a width of 0.1 object space units. Both look reasonable. The fragment shader will also compute the radius of the ring based on the contact time, the current time (`world_time`), and the speed. The width will be used to compute an outer radius. If the distance from the fragment to the contact point is between these two radii then the fragment shader will emit a red color, otherwise it will emit the input color.

Two aspects of the solution were chosen to reduce the amount of work performed by the rasterizer and fragment shader. Since the fragment shader needs the eye-space vertex coordinate for lighting, it would make sense to transform the contact position to eye space *in the geometry shader* and then send it to the fragment shader. (Remember that for each primitive emitted by the geometry shader there can be dozens or more fragments.) Note that the contact position applies to the entire tile. It is being emitted as a vertex attribute (named `tact_pos_time_e`), but the same value is emitted for all four vertices making up the tile's triangle strip. It would be a waste of effort for the rasterizer to interpolate the values. For that reason the interface block declares `tact_pos_time_e` as `flat`.
Geometry Shader: The geometry shader given in the solution to Problem 2 converts the contact position to eye space and emits it, along with the time, in a new geometry shader output `tact_pos_time_e`. The interface block appears below, see the Problem 2 solution for the geometry shader code.

```
in Data_to_FS {
  flat vec3 normal_e;
  vec4 vertex_e;
  flat vec4 color;

  // Pass contact position and time to fragment shader to draw the ring.
  flat vec4 tact_pos_time_e;
};
```

Fragment Shader:

```
void fs_main_tiles_2() {
  float age = world_time - tact_pos_time_e.w;

  float speed = 0.5;   // Speed of increase of the ring radius.
  float width = 0.1;   // Width of ring.

  // Compute distance from this fragment to the contact point.
  float dist = distance(vertex_e.xyz,tact_pos_time_e.xyz);

  float inner_radius = age * speed;
  float outer_radius = inner_radius + width;

  // Determine if this fragment is in the ring, and set color appropriately.
  bool ring = dist >= inner_radius && dist <= outer_radius;
  vec4 color_red = vec4(1,0,0,1);
  vec4 color2 = ring ? color_red : color;

  gl_FragColor = generic_lighting(vertex_e, color2, normal_e);
  gl_FragDepth = gl_FragCoord.z;
}
```