Project 4 : Voronoi Diagram Mosaics

1 Overview

OpenGL can be used not only for 3D drawing, but also for solving various geometric problems. Here, we will use it to approximate the *Voronoi diagram* of a collection of points in the plane.

Assume you are given a finite number of points, $p_1, p_2, \ldots, p_n$. Let's call them *sites*. For each site, there is an area around it (called Voronoi region) which consists of all points for which that site is the closest of all sites. If you are somewhere on the plane and want to get to some site as quickly as possible, you would want to walk to the site whose Voronoi region you are in, since it's the closest one.

Approximate Voronoi diagrams can be drawn using OpenGL as follows. For each site, plant a cone at that site. Assuming the plane is horizontal (think of it as the ground) the cones grow perpendicularly up. The angles of the cones are all the same. Assign different colors to the cones and draw them, using a parallel projection in the direction perpendicular to the ground. When drawing, use no illumination, just the cones' colors. Thus, the color of a pixel will be the same as the color of the cone that can be seen from that pixel when looking (vertically) up.

In this project, you will implement Voronoi diagram computation in OpenGL, that is, set up the projection as described above, and draw the cones in different colors. To make it more fun, the colors for the cones could be either random or selected from a ppm file (you can hardcode the name to be '`input.ppm`'). If you pick color from an image, the output looks like a mosaic with irregular tiles.

2 Requirements

Your code should start by showing a Voronoi diagram of 32 randomly chosen (all within the viewport) sites. Choose the colors for the sites randomly (say, from a uniform distribution on rgb values).

Program the following menu items:

- **Spray more points.** Increase the number of sites by a factor of two (e.g. if we choose this menu item seven times, we would like to see a Voronoi diagram of 4096 sites). Since the algorithm is computationally expensive, limit the number of sites to $65,536$.

- **Show/hide sites.** This one should toggle between two modes: sites visible (drawn as black dots) and sites invisible (not drawn at all, just their Voronoi regions visible). Initially, show the sites.

- **Reset.** Go back to the first setting: 32 sites, random coloring of Voronoi regions, no movement, sites shown as black dots.

- **Move points.** This should cause the sites to move along circular trajectories around the center of the window. Choose the angular velocity of each point randomly (make the velocity of each site different). Try to experiment with velocity so that things look nicely.

- **Toggle coloring method.** There are two coloring methods we'll use. The first is just random colors for each Voronoi region. The second is colors looked up from a square ppm image without any comment lines. Read it in using the `createRGBTexture2D` function provided in the sample code for project 3. You don't need to implement point movement in the image coloring mode (it's OK if the points stop or move to the original locations if this mode is selected). When the sites are moving in the random coloring mode, keep the cone's color constant for each of the moving sites. This will make it easy to see how the cells evolve as the sites move.

3 Grading

1. Voronoi diagrams look OK, 40

2. Animation 30

3. Mosaic 30

4 Random values

Use `drand48` or other random number generator to produce site coordinates, colors and velocities **on the CPU** and send the results to buffers on the GPU in the same way as in the past projects. Random number generation is tricky on the GPU because of it typically maintains and runs hundreds or thousands threads in parallel, and random number generators are notoriously hard to parallelize. There are algebraic tricks that have been reported to work well (try googling random numbers glsl), but the quality of the resulting random numbers may be insufficient.

`drand48` is convenient since it produces a random double precision floating point number in [0,1]. To produce a random number in $[-s, s]$, you can simply do $s * (1 - 2 * \texttt{drand48}())$. For point coordinates, you'll probably want to use range $[-1, 1]$ for their x- and y- coordinates. For colors, you can use the range $[0, 1]$, which will give you values that can readily be written into the output of the fragment shader (in OpenGL4). For angular velocities, experiment to find a good range $[-s, s]$.

It is probably easiest to simply generate the data (coordinates, colors and velocities) for all 65536 sites that one may potentially want to use at startup and send it to the buffer(s). Then, you would render only the requested number of cones by using only a part of the buffer(s).

5 Color lookup

In the image lookup mode, use texture to lookup colors. The texture coordinates you need are basically the location of the site scaled to $[0, 1]$.

6 Cones

A new convenient trick you may want to use is to alter the depth of a fragment in the fragment shader. This will allow you to draw cones without explicitly modeling them using triangles.

To do that, use the built-in `gl_FragDepth` output variable (which is of type float). There is no need to declare it (since it's built in). To assign a new depth to a fragment, use

```
gl_FragDepth = new_depth;
```

To draw a cone, draw a shape that covers the entire window, e.g. quad, or a large triangle, for each of your sites. You may want to send the coordinates of that quad/triangle as output variables to the fragment shader to have the fragment coordinates (in $[-1, 1] \times [-1, 1]$)) available and therefore simplify the distance calculation described below.

In the fragment shader, make the depth proportional to the distance between site's coordinates and the fragment's coordinates. GLSL provides several functions to support vector calculations. For this project, `length` will be useful. You can use it to compute length of a vector, or distance between points. For example, if `a` and `b` are of type `vec2`, then `length(a-b)` returns the distance between `a` and `b`. `distance(a,b)` should work as well. The valid range for values written into `gl_FragDepth` is $[0, 1]$. In our case, it's enough to divide distance by 3 to make it fit into $[0, 1]$ (3 is less than the diameter of the square extending from -1 to 1 in x and y).

6.1 Showing and hiding sites

To show the sites' locations, simply set the color of the fragment to black (i.e. $(0, 0, 0)$) if its distance to the site (computed as described earlier) is less than a certain small number. Pick that number so that the sites appear as disks no more than a few pixels wide in the window of size used in the sample code (start off around 0.005).

7 Projection, modelview transformation etc.

In this project, you don't really need them. Basically, render using the canonical projection (see first slides of the projections set). The matrix of the canonical projection is the identity, and the view volume extends from -1 to 1 in the x- and y- directions and from 0 to 1 in the z-direction. For each site, you can simply render the square with vertices $(\pm 1, \pm 1)$, for example using two triangles - its projection precisely covers your OpenGL window. For a vertex at, say, (x, y), you want to assign $(x, y, 0, 1)$ to the `gl_Position` variable in the vertex shader. It is important to use 1 as the homogeonous coordinate - in particular, don't use zero since that would mean vertex at infinity.

8 Instanced rendering

A unique aspect of this project is that what needs to be done is to render a number of squares that are then turned into cones using the depth assignment on the fragment processing stage. In order to do that, we need to send identical geometry into the pipeline several times (in OpenGL jargon: render several instances of the same object), with attributes (color of the cone, location of the site, angular velocity) that depend only on the instance rather than on a particular vertex (as in projects 2 and 3).

Instanced rendering is the elegant way to go in such cases. Instanced rendering uses a single OpenGL function to send several copies (called instances) of a vertex stream into the pipeline. Instanced rendering is generally more efficient than a loop in the CPU code that sends the same vertex stream several times, since it is easier for the driver to optimize the rendering (e.g. it knows that there is no way to change the OpenGL state between instances but it cannot make the same assumption if a CPU loop is used, at least not without careful analysis of the CPU code). Instanced rendering is also better an faster than using a large buffer with repeated data (which is an obvious alternative). First, you save memory. Second, GPU does not need to use as much memory bandwidth and has more opportunities to cache the vertex data in faster memory if it has it.

In instanced rendering, you generally have two types of vertex attributes: instanced ones and regular ones (non-instanced). Both map to input variables of the vertex shader and are specified using buffers that we are used to. However, the way the value of an input variable is looked up from the respective buffer is different for the two attribute types.

For a regular input variable, the value for the i-th vertex *for each instance* is looked up from the respective buffer using index i.

For an instanced attribute, the value for *each vertex* of the i-th instance is looked up from the respective buffer using index i/d, where d is an positive integer called the *divisor* of the attribute and / stands for *integer* division.

You want to use instanced attributes for attributes that should be the same for all vertices in the same instance. In the project, these would be site location, color (used in random coloring mode) and angular velocity. If it is needed to transmit the values of these attributes to the fragment shader, you typically want to use the flat interpolation since it is fastest and the values are the same for all triangles anyway.

Regular attributes are ones that don't depend on instance. In the project, the locations of the vertices of the square covering the window are natural regular attributes.

Other than that, everything works in the way we are used to.

8.1 Extensions provided by the sample code to support instanced rendering

Recall that to associate a buffer `b` (of type `Buffer*`) with attribute of index `i` in a vertex array `a` (of type `VertexArray*`) we used the `attachAttribute` method:

```
a->attachAttribute(i,b);
```

We extended this method to allow 3 arguments, with the last one being the divisor, for instanced attributes. So, if you want to make the attribute in buffer `b` instanced with the divisor of 1 (i.e. advance the lookup index by 1 for each new instance - this is what you want in the project), use

```
a->attachAttribute(i,b,1);
```

Note that the two-argument version of `attachAttribute` is still there and you should use it for the regular attributes (in the project: locations of the vertices of the square, $(\pm 1, \pm 1)$).

We also extended functions for sending vertex streams into the pipeline to allow instanced rendering. The methods are the same as before, `sendToPipeline` and `sendToPipelineIndexed`, but they allow an additional integer argument representing the number of instances you would like to send. For example, if you do

`a->sendToPipeline(GL_TRIANGLE_STRIP,0,4,3)`,

you are sending the first four vertices with indices $0 \ldots 3$ '3 times'. Why quotes around 3 times? Because attributes are looked up from buffers as described above, the attribute values may be different for any two vertices: regular ones are repeated for each instance and instanced ones are constant for all vertices within an instance but vary according to instance number. This is why you'll be able to draw a cone with a different apex and a different color for each instance.

Similarly, there is a 5-argument version of the `sendToPipeline`, with the last argument being the desired number of instances.