Circle Modeling

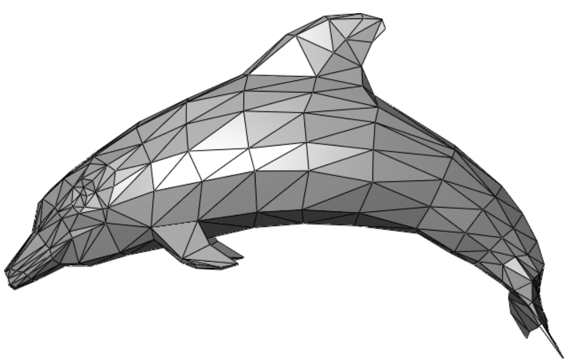
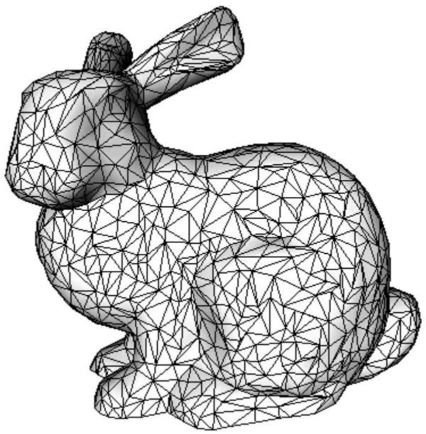
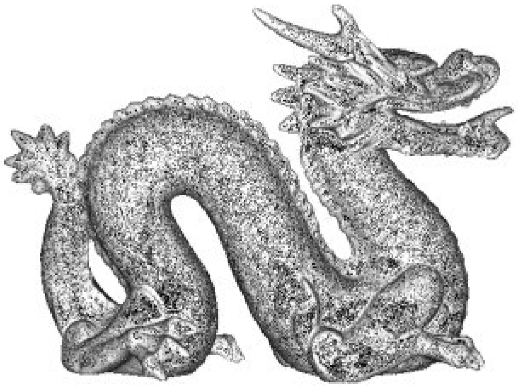
Computer Graphics
Instructor: Sungkil Lee

Background: Geometric Models

Models

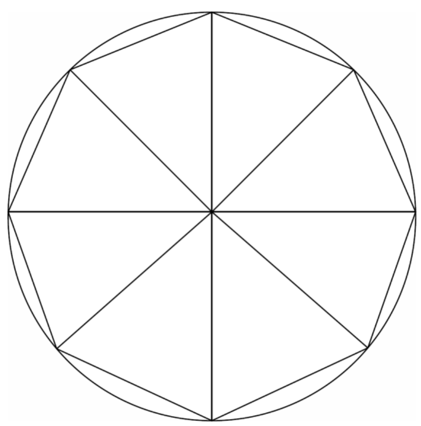
- **Models:**
  - Mathematical abstraction of the real world or virtual worlds.

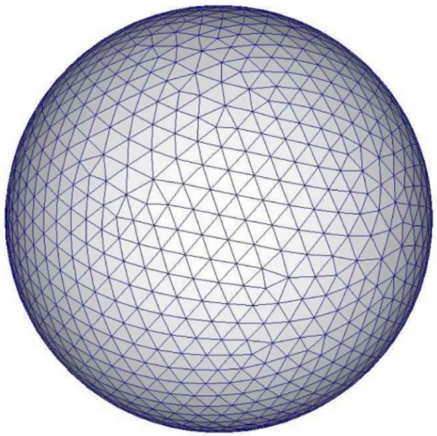
- **Geometric Models:**
  - In CG, we model our worlds with *geometric objects*.
  - Building blocks: a set of simple 3D primitives (points, lines, triangles, ...)
  - ***Triangular meshes*** are common, which comprises a set of triangles connected by their common edges or corners.

3D Primitives

- **3D objects that fit well with graphics HW and SW:**
  - described by their 2D surfaces and can be thought of as being hollow.
    - c.f., objects with 3D surfaces are called the *volumetric* objects (e.g., CT).
  - can be specified through a set of vertices.
  - either are composed of or can be approximated by flat, convex polygons.
    - e.g., a circle/sphere approximated by flat triangles.





3D Primitives

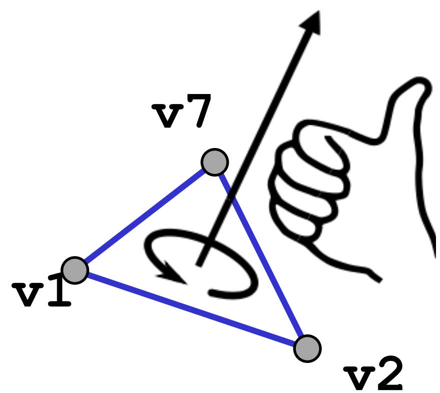
- **Why we set these conditions?**
  - Modern graphics systems are optimized for rendering **triangles** or **meshes of triangles** (e.g., more than 100 M triangles / sec.).
    - Points and lines are also supported well.
  - Vertices can be processed with the pipeline architecture, independently.

- **Why are triangles fundamental primitives?**
  - *The triangles are always flat.*
  - General polygons might not lie in the same plane, and then, there is no simple way to define interior of the object.
  - Also, general polygons can be decomposed into a set of triangles:
    - then, we can apply the same pipeline on the triangles.

More on State Setup

Vertex Ordering for a Triangle

- **In general, triangles are not double-sided.**
  - Hence, we need to set the direction of a triangle face.
  - In OpenGL, we use the order of vertices to distinguish front-facing vs. back-facing triangles.
    - Counter-clockwise encirclement of outward-pointing normal.



  - The order {v1, v2, v7} and {v2, v7, v1} (front-facing) define the same polygon with the same face direction, but the order {v1, v7, v2} (back-facing) is different.

Back-Face Culling

- **By default, OpenGL will render back-facing triangles as well as front-facing triangles.**
  - You need to explicitly command not to render back-facing triangles.

```
glEnable( GL_CULL_FACE );      // enable face culling
glCullFace( GL_BACK );         // cull back faces
glFrontFace( GL_CCW );         // counterclockwise encirclement determine
                               // the direction of a face.
```

  - This mechanism is called the *back-face culling*.
  - You can query the current state of the face culling as follows.

```
glIsEnabled( GL_CULL_FACE );
```

Wireframe mode rendering

- **Wireframe mode (desktop only):**
  - To see how triangles are organized, we can turn on the wireframe mode.
  - Set glPolygonMode as GL_LINE for wireframe mode or GL_FILL for solid mode.
  - You can change the line width using glLineWidth().

```
void keyboard( GLFWwindow* window, int key, int scancode, int action, int mods )
{
   ...
   else if(key==GLFW_KEY_W)
   {
      bWireframe = !bWireframe;
      glPolygonMode( GL_FRONT_AND_BACK, bWireframe ? GL_LINE:GL_FILL );
      printf( "> using %s mode\n", bWireframe ? "wireframe" : "solid" );
   }
   ...
}
```

Definition of Geometry

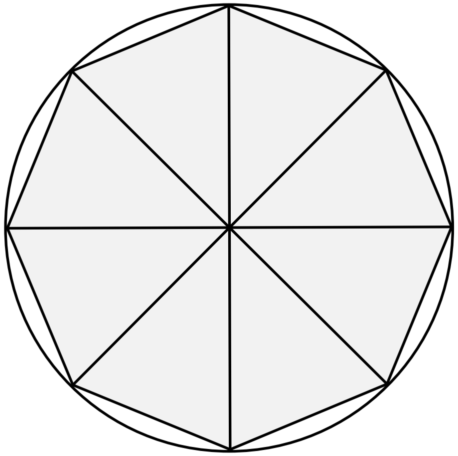
Where you make a geometry

- **It actually does not matter.**
  - as long as OpenGL stuffs are initialized.
  - However, it is clean and easy to do it in `user_init()`.
  - This is because we usually create geometric objects only once.

```
// usually called after basic GL stuffs are initialized
void user_init()
{
	... // create objects here ...
}
```

Circle Definition

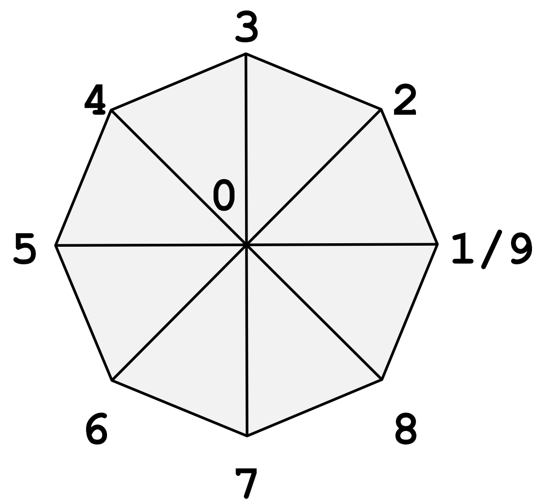
- **Polygonal approximation of a circle**
  - Modern OpenGL supports only triangles as polygonal primitives.
    - Implicit curves, such as a circle, are not supported.
    - That is, we cannot use $x^2 + y^2 = r^2$ for drawing.
  - We thus need to approximate a circle using a finite set of triangles.
  - As we increase the number of triangles, the shape becomes close to circle.



An octagonal approximation of a circle

Circle Definition

- **Definition of vertex indices of vertices**



- **Polar coordinates of vertices**
  - k-th boundary vertex of N-gon of a radius one has the following polar coordinates:

$$(x, y) = \left(\cos\frac{2\pi}{N} \times k, \sin\frac{2\pi}{N} \times k \right)$$

Circle Definition

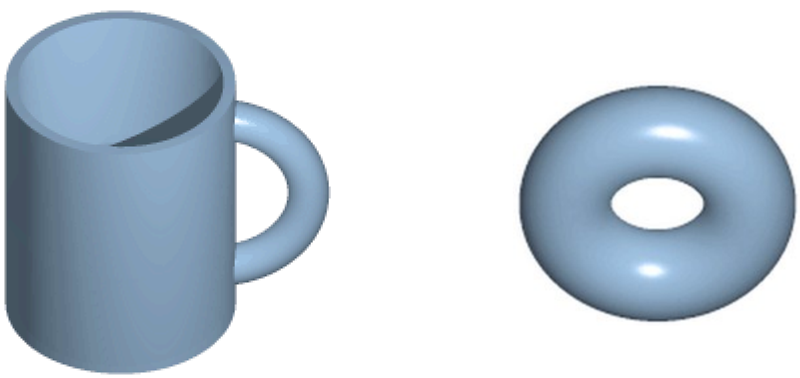
- **Define arrays for vertices**
  - Be sure that the positions here are defined in LHS form of the canonical view volume; z-axis goes farther from the eye.
    - This 2D circle is actually defined in 3D (z=0)
    - When you model 3D objects, pay more attention to the z axis.

```
std::vector<vertex> create_circle_vertices( uint N )
{
   std::vector<vertex> v = {{ vec3(0), vec3(0,0,-1.0f), vec2(0.5f) }}; // origin
   for( uint k=0; k <= N; k++ )
   {
      float t = PI*2.0f*k/float(N), c=cos(t), s=sin(t);
      v.push_back
      ({
         vec3(c,s,0),            // vertex position
         vec3(0,0,-1.0f),        // normal vector facing your eye
         vec2(c,s)*0.5f+0.5f     // texture coordinate in ([0,1], [0,1])
      });
   }
   return v;
}
```

Object Specification

Geometry vs. Topology

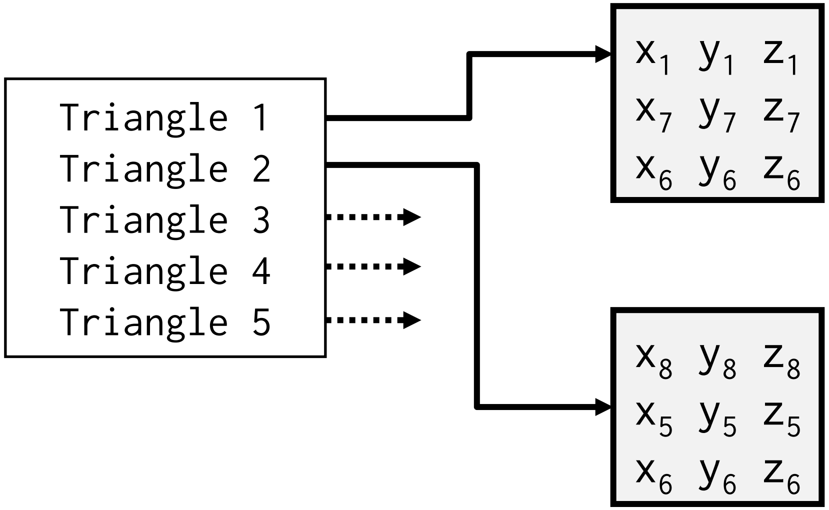
- **Generally, it is a good idea to look for data structures that separate the geometry from the topology**
  - *Geometry*: locations of the vertices
  - *Topology*: structural organization of the vertices and edges
    - Connectedness is preserved under continuous deformation
    - Topology holds even if geometry changes



The cup and torus share the same topology.

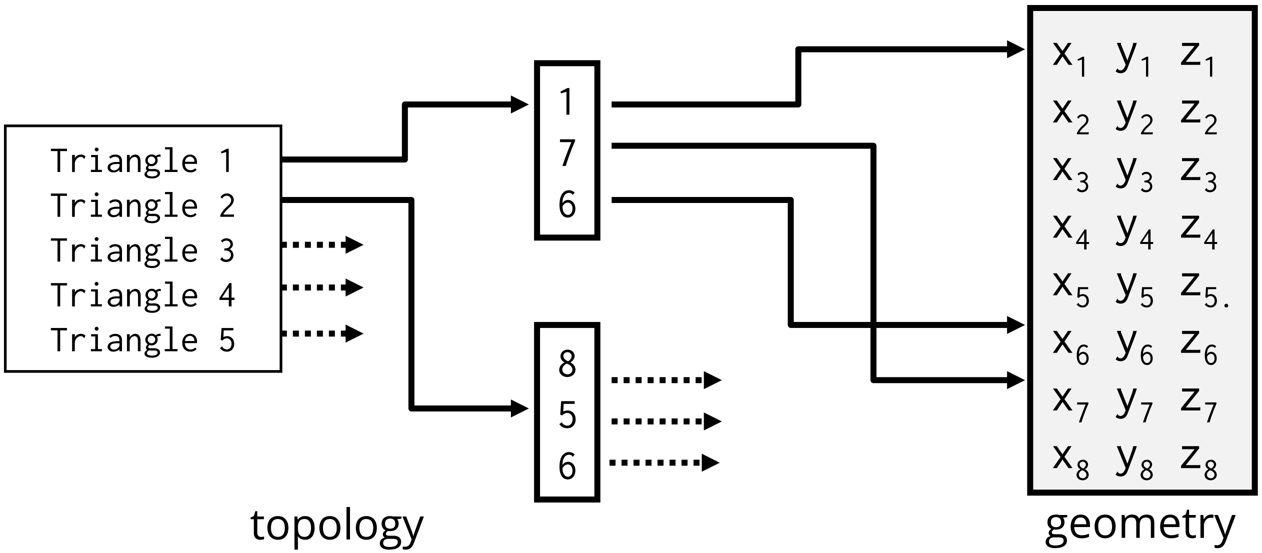
Method 1: Simple Vertex Buffering

- **A single vertex buffer defines geometry and topology.**
  - Topology information is hard-coded in a vertex buffer.
  - When a vertex moves to a new location, we must search and replace it for all the occurrences.
  - Often inefficient and unstructured.



Method 2: Index Buffering

- **Using vertex buffer + index buffer together**
  - **Topology** is separated from geometry by indexing scheme.
  - Use ***indices*** from the vertices into this array.



- **Typically faster than simple vertex buffering**
  - Index buffering avoids redundant vertex shading, while the simple vertex-only buffering has duplicate vertices in its definition.

Simple Vertex-Only Buffering

Simple Vertex Buffering

- **For an N-gon, we need N×3 vertices.**
  - Pay attention to make out-facing triangles (counter-clockwise order)

```
void update_vertex_buffer( const std::vector<vertex>& vertices, uint N )
{
   ...

   std::vector<vertex> v; // triangle vertices
   for( uint k=0; k < N; k++ )
   {
      v.push_back(vertices.front());  // the origin
      v.push_back(vertices[k+1]);
      v.push_back(vertices[k+2]);
   }

   // generation of vertex buffer: use triangle_vertices instead of vertex_list
   glGenBuffers( 1, &vertex_buffer );
   glBindBuffer( GL_ARRAY_BUFFER, vertex_buffer );
   glBufferData( GL_ARRAY_BUFFER, sizeof(vertex)*v.size(), &v[0], GL_STATIC_DRAW );
}
```

Simple Vertex Buffering

- **render()**
  - Render N×3 vertices instead of 3 vertices in the hello example.

```
void render()
{
    ...

    // render vertices: trigger shader programs to process vertex data
    glDrawArrays( GL_TRIANGLES, 0, NUM_TESS*3 ); // NUM_TESS = N

    ...
}
```

Index Buffering

Index Buffering

- **Index definition**
  - We only specify the **topology** for indices.
  - Use the vertex buffer array (**for geometry**) as it is.
  - We use N×3 indices unlike the simple vertex buffering.

```
void update_vertex_buffer( const std::vector<vertex>& vertices, uint N )
{
   ...

   indices.clear();
   for( uint k=0; k < N; k++ )
   {
      indices.push_back(0); // the origin
      indices.push_back(k+1);
      indices.push_back(k+2);
   }
}
```

Index Buffering

- **Vertex/index buffer definition**
  - We need two buffers, vertex buffer and index buffer, simultaneously.
  - The index buffer uses **GL_ELEMENT_ARRAY_BUFFER** as a buffer type.
  - Vertex buffer will use the initial vertices directly (without connectivity).

```
void update_vertex_buffer( const std::vector<vertex>& vertices, uint N )
{
  ...

  // generation of vertex buffer: use vertex_list as it is
  glGenBuffers( 1, &vertex_buffer );
  glBindBuffer( GL_ARRAY_BUFFER, vertex_buffer );
  glBufferData( GL_ARRAY_BUFFER, sizeof(vertex)*vertics.size(), &vertices[0], GL_STATIC_DRAW);

  // geneation of index buffer
  glGenBuffers( 1, &index_buffer );
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,index_buffer);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(uint)*indices.size(),&indices[0],GL_STATIC_DRAW);
}
```

Index Buffering

- **Vertex array definition with index buffering**
  - Unlike the simple vertex-only buffering, we also provide the index buffer as input **cg_create_vertex_array**().
  - When you bind the vertex array, the vertex and index buffers and their binding are bound at the same time.

```
void update_vertex_buffer( const std::vector<vertex>& vertices, uint N )
{
   ...

   // generate vertex array object, which is mandatory for OpenGL 3.3 and higher
   if(vertex_array) glDeleteVertexArrays(1,&vertex_array);
   vertex_array = cg_create_vertex_array( vertex_buffer, index_buffer );
}
```

Index Buffering

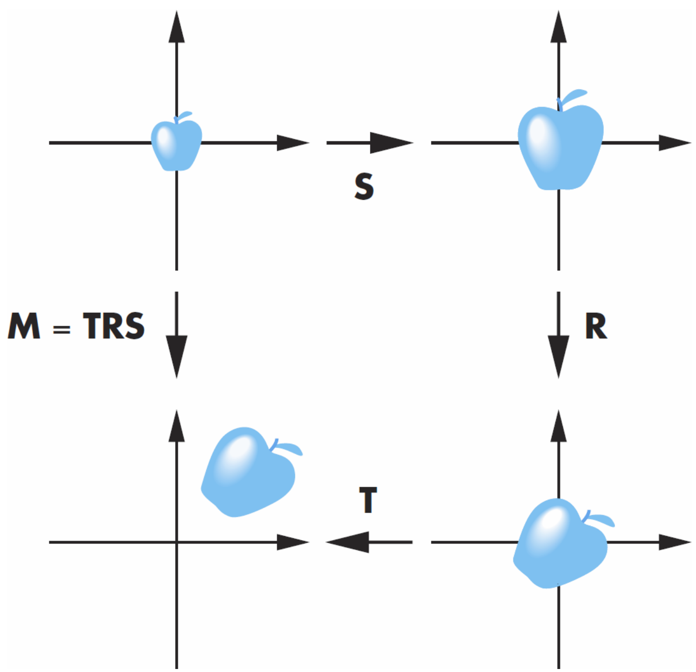
- **render()**
  - Binding the vertex array handles the binding of index buffers as well.
  - Render N×3 indices instead of N×3 vertices in the simple vertex buffering.
  - Use `glDrawElements`() instead of `glDrawArrays`() to use the index buffering.

```
void render()
{
    ...
    glBindVertexArray( vertex_array );

    ...
    glDrawElements( GL_TRIANGLES, NUM_TESS*3, GL_UNSIGNED_INT, nullptr );

    ...
}
```

Instancing

Instancing

- **In modeling, we often start with an object centered at the origin, oriented with the axis, and at a standard size.**
  - We apply an ***instance transformation*** to its vertices to scale, orient, and locate somewhere.
  - This allows us to work with minimal geometric objects, while rendering many different objects.



Instancing

- **To realize the concept of instancing, we use a unit vertex buffer:**
  - We create a single vertex buffer, which is unit-sized and located at the origin.
  - In render(), we use a loop to render multiple objects.
  - In the loop, we change the size and position for each circle, and pass them to their uniform variables residing in (vertex or fragment) shaders.

- **Refer to the circle example in the following pages.**

Example: Drawing Two Circles

- **First, define the structure of objects.**
  - Here, we define a circle structure.
  - The attributes include the center position, radius, rotation angle, color, and modeling matrix.
  - We also define an update() function for per-circle updates.

```
// in circle.h

struct circle_t
{
   vec2  center=vec2(0);   // 2D position for translation
   float radius=1.0f;      // radius
   float theta=0.0f;       // rotation angle
   vec4  color;            // RGBA color in [0,1]
   mat4  model_matrix;     // modeling transformation

   // public functions
   void  update( float t );
};
```

Example: Drawing Two Circles

- **create_circles() instantiates many circles.**
  - Here, two circle objects are instantiated.

```
std::vector<circle_t> create_circles()
{
   std::vector<circle_t> circles;
   circle_t c;

   c = {vec2(-0.5f,0),1.0f,0.0f,vec4(1.0f,0.5f,0.5f,1.0f)};
   circles.emplace_back(c);

   c = {vec2(+0.5f,0),1.0f,0.0f,vec4(0.5f,1.0f,1.0f,1.0f)};
   circles.emplace_back(c);

   return circles;
}
```

Example: Drawing Two Circles

- **circle_t::update() builds a transformation matrix.**
  - radius and theta are user-defined parameters for animation.
  - The parameters are used to build a 2D transformation matrix.
  - The details will be explained later in the transformation lecture.

```
void circle_t::update( float t )
{
   radius   = 0.35f+cos(t)*0.1f;    // simple animation
   theta    = t;
   float c  = cos(theta), s=sin(theta);

   // these transformations will be explained in later transformation lecture
   mat4 scale_matrix = { radius,0,0,0,0,radius,0,0,0,0,1,0,0,0,0,1 };
   mat4 rotation_matrix = { c,-s,0,0,s,c,0,0,0,0,1,0,0,0,0,1 };
   mat4 translate_matrix = { 1,0,0,center.x,0,1,0,center.y,0,0,1,0,0,0,0,1};

   model_matrix = translate_matrix*rotation_matrix*scale_matrix;
}
```

Example: Drawing Two Circles

- **In render(), update per-circle parameters and matrices.**
  - Here, we change the color and matrix for each circle.
  - Then, we call glDrawElements for each circle, repeatedly.
  - Your shader draws them differently, based on the different uniforms.

```
void render(){
   ...
   for( auto& c : circles )
   {
      c.update(t); // per-circle update

      // update per-circle uniforms
      GLint uloc;
      uloc = glGetUniformLocation( program, "solid_color" );
      glUniform4fv( uloc, 1, c.color );   // pointer version
      uloc = glGetUniformLocation( program, "model_matrix" );
      glUniformMatrix4fv( uloc, 1, GL_TRUE, c.model_matrix );

      // per-circle draw calls
      glDrawElements( GL_TRIANGLES, NUM_TESS*3, GL_UNSIGNED_INT, nullptr );
   }
```

Vertex and Fragment Shaders

Your vertex shader

- **In the vertex shader, we locate the vertices based on its transformation and attributes.**
  - Here, we apply scaling, rotation, and translation in a row.
  - We first scale the vertex with the circle radius, and rotate it.
  - Then, we add its offset, which is the center of the circle.

```
...

// uniform variables
uniform mat4   model_matrix;   // 4x4 transformation matrix
uniform mat4   aspect_matrix; // tricky 4x4 aspect-correction matrix

void main()
{
   gl_Position = aspect_matrix * model_matrix * vec4(position,1);
   ...
}
```

Your vertex shader

- **One last stuff to do is the correction of the aspect ratio.**
  - Since we specify the vertex position in the default viewing volume, the resulting shape in the horizontally/vertically wider screen will be distorted.
  - To handle this, using matrix is consistent in vertex shader.

```
void update()
{
   ...

   // tricky aspect correction matrix for non-square window
   float aspect = window_size.x/float(window_size.y);
   mat4 aspect_matrix = { min(1/aspect,1.0f),0,0,0,0,min(aspect,1.0f),0,0,
      0,0,1,0,0,0,0,1 };

   // update common uniform variables in vertex/fragment shaders
   GLint uloc = glGetUniformLocation( program, "aspect_matrix" );
   if(uloc>-1) glUniformMatrix4fv( uloc, 1, GL_TRUE, aspect_matrix );
   ...
}
```

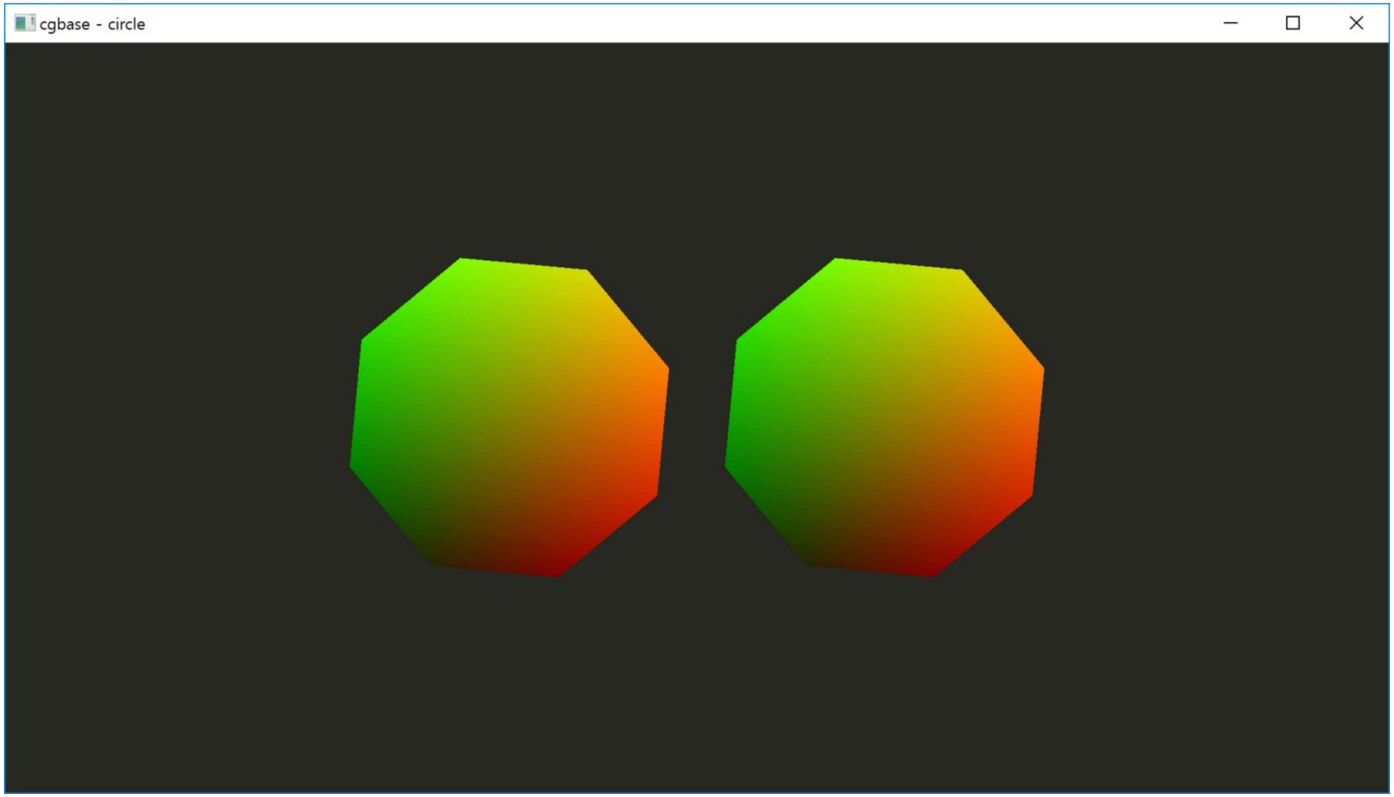
Your fragment shader

- **Nearly the same as the hello example.**
  - Additionally, we visualize texture coordinates as color output.

```
// inputs from vertex shader
in vec2 tc; // used for texture coordinate visualization

...

void main()
{
   fragColor = b_solid_color ? solid_color : vec4(tc.xy,0,1);
}
```

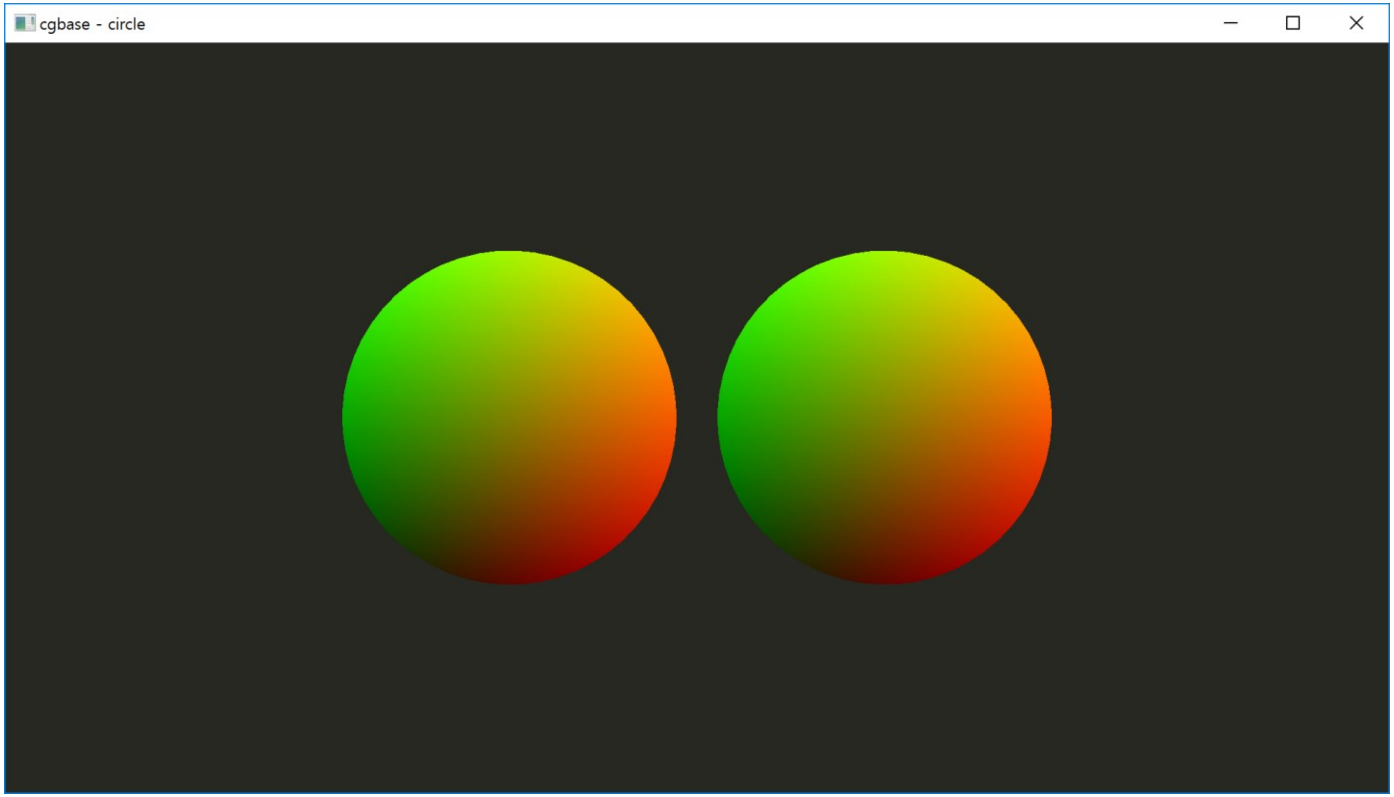
Results

- **Octagonal approximation**
  - Color indicates the texture coordinates.



Results

- **64-gon approximation**
  - Now, they looks almost like circles.



Results

- **Wireframe-mode rendering (not supported in OpenGL ES)**
  - Now, you can see the triangular structure.