Programming with WebGL Shaders

Sorin Babii
sorin.babii@cs.upt.ro

Objectives

- Simple Shaders
  - Vertex shader
  - Fragment shaders
- Programming shaders with GLSL
- Finish first program

Vertex Shader: Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

Fragment Shader: Applications

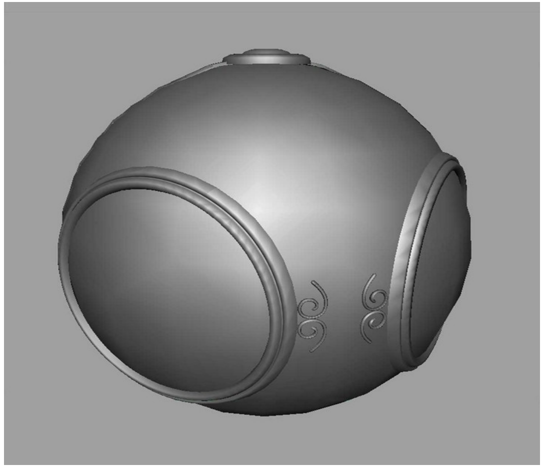
Per fragment lighting calculations

per vertex lighting

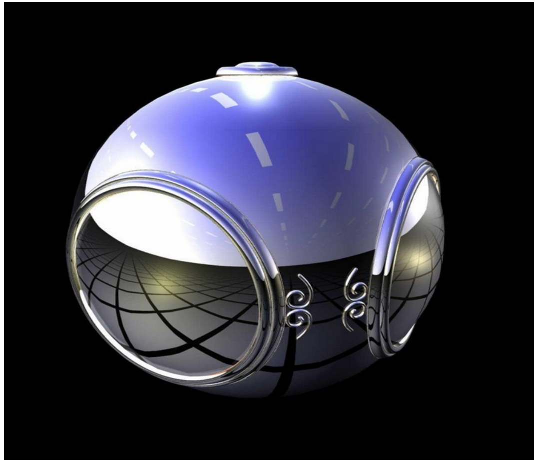
per fragment lighting

Fragment Shader: Applications

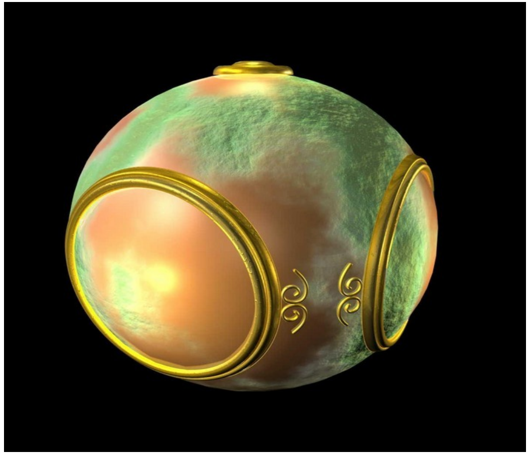
Texture mapping



smooth shading



environment mapping



bump mapping

Writing Shaders

- First programmable shaders - more like an assembly language
- OpenGL extensions added functions for vertex and fragment shaders
- Cg (C for graphics) C-like language for programming shaders
  - Works with both OpenGL and DirectX
  - Complex Interface to OpenGL
- OpenGL Shading Language (GLSL)

GLSL

- OpenGL Shading Language
- Part of OpenGL 2.0 and up ⇒ not in OpenGL ES 1.x ☹
- High level C-like language
- New data types
    - Matrices
    - Vectors
    - Samplers
- As of OpenGL 3.1, application must provide shaders

Simple Vertex Shader

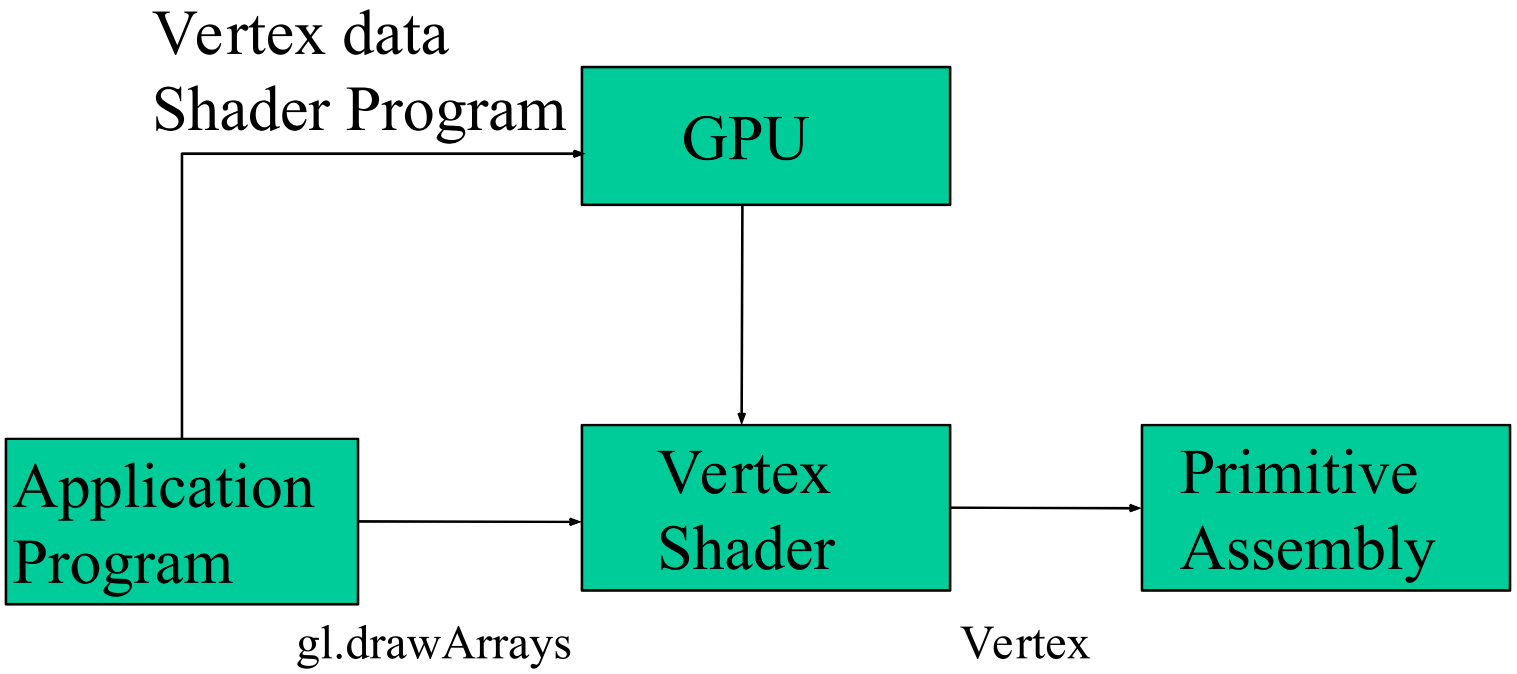
input from application

```
attribute vec4 vPosition;

void main(void)

{

   gl_Position = vPosition;

}
```

must link to variable in application

built-in variable

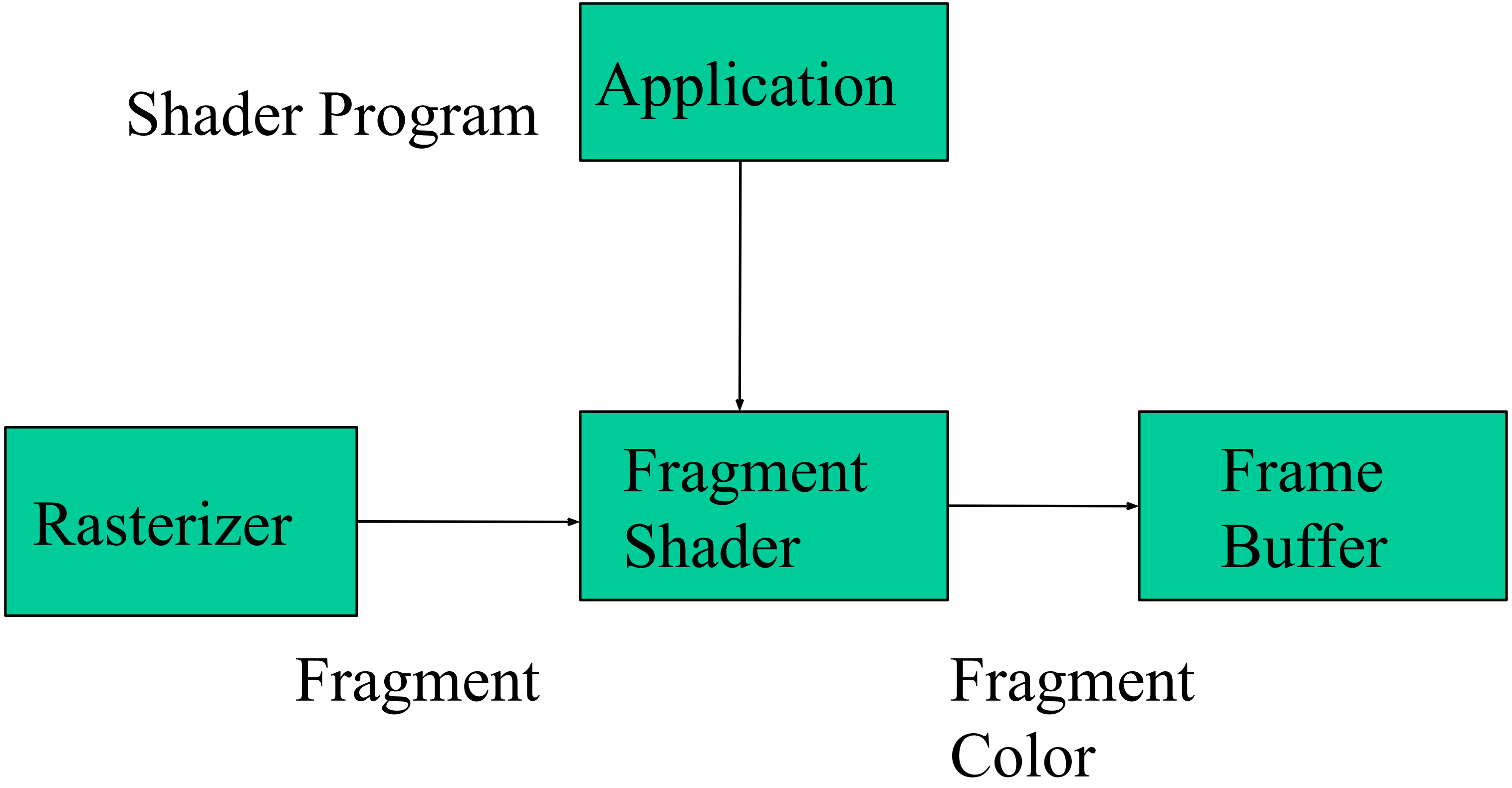
Execution Model



Simple Fragment Program

```
precision mediump float;

void main(void)

{

   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);

}
```

Execution Model



Programming with WebGL Shaders (cont.)

Sorin Babii
sorin.babii@cs.upt.ro

Data Types

- C types: int, float, bool
- Vectors:
    - -float vec2, vec3, vec4
    - -Also int (ivec) and boolean (bvec)
- Matrices: mat2, mat3, mat4
    - -Stored by columns
    - -Standard referencing m[row][column]
- C++ style constructors
    - -vec3 a =vec3(1.0, 2.0, 3.0)
    - -vec2 b = vec2(a)

No Pointers

- There are no pointers in GLSL
- We can use C structs which can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.

```
mat3 func(mat3 a)
```

- variables passed by copying

Qualifiers

- GLSL has many of the same qualifiers such as `const` in C/C++
- Need others due to the nature of the execution model
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
- There are a few built in variables such as `gl_Position` but most have been deprecated
- User defined (in application program)

```
attribute float temperature
attribute vec3 velocity
```

recent versions of GLSL use `in` and `out` qualifiers to get to and from shaders

Uniform Qualified

- Variables that are constant for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the time or a bounding box of a primitive or transformation matrices

Varying Qualified

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- With WebGL, GLSL uses the varying qualifier in both shaders

```
varying vec4 color;
```

- More recent versions of WebGL use out in vertex shader and in in the fragment shader

```
out vec4 color; //vertex shader
in vec4 color;  // fragment shader
```

One Naming Convention

- attributes passed to vertex shader have names beginning with v (v Position, vColor) in both the application and the shader
  - -Note that these are different entities with the same name
- Variable variables begin with f (fColor) in both shaders
  - -must have same name
- Uniform variables can have the same name in application and shaders

Example: Vertex Shader

```
attribute vec4 vColor;
varying vec4 fColor;
void main()
{
  gl_Position = vPosition;
  fColor = vColor;
}
```

Corresponding Fragment Shader

```
precision mediump float;

varying vec3 fColor;
void main()
{
  gl_FragColor = fColor;
}
```

Sending Colors from Application

```
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors),
                    gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );
```

Sending an Uniform Variable

```
// in application

vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
colorLoc = gl.getUniformLocation( program, ”color" );
gl.uniform4f( colorLoc, color);

// in fragment shader (similar in vertex shader)

uniform vec4 color;

void main()
{
    gl_FragColor = color;
}
```

Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```

Arrays

- Can refer to array elements by element using [] or selection (.) operator with
  - -x, y, z, w
  - -r, g, b, a
  - -s, t, p, q
  - –a[2], a.b, a.z, a.p are the same
- Manipulating components

```
vec4 a, b;
a.yz = vec2(1.0, 2.0, 3.0, 4.0);
b = a.yxzw;
```

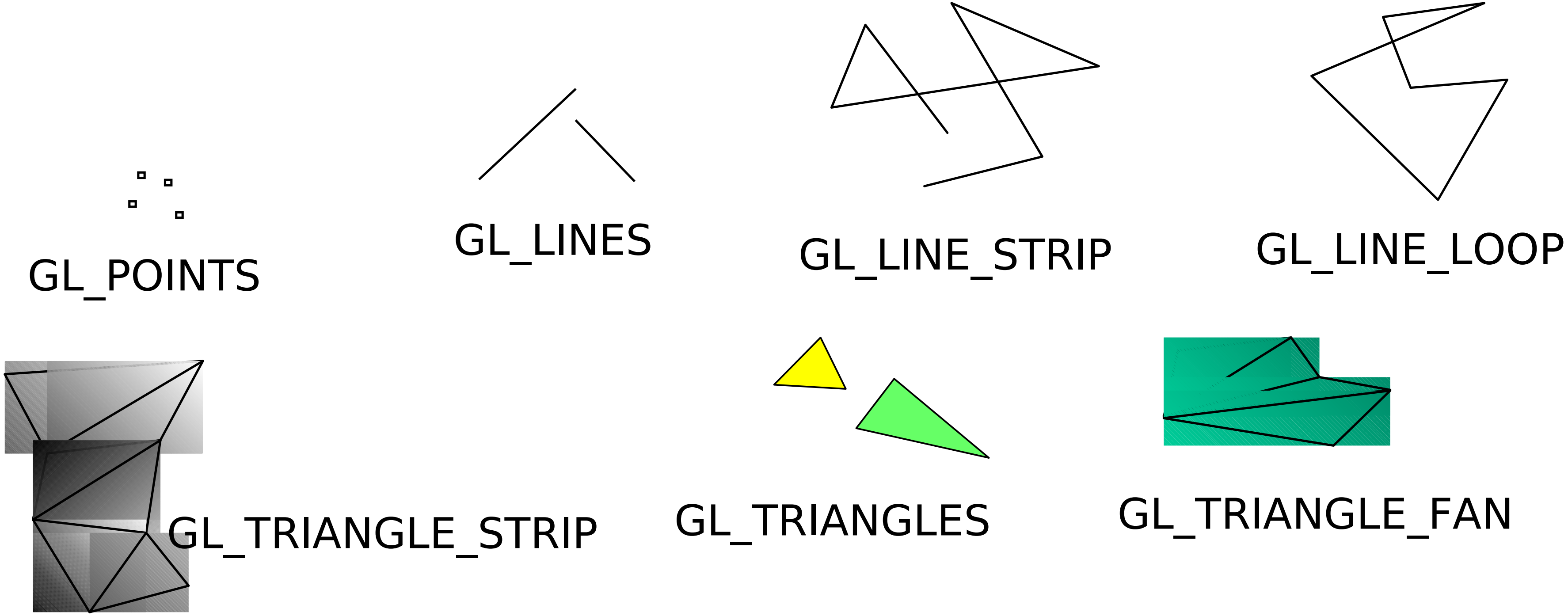
Programming with WebGL
Color and Attributes

Sorin Babii
sorin.babii@cs.upt.ro

Objectives

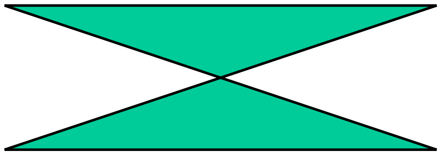
- Expanding primitive set
- Adding color
- Vertex attributes

WebGLPrimitives

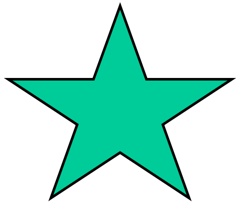


Polygons

- WebGL will only display triangles
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator but not WebGL



nonsimple polygon



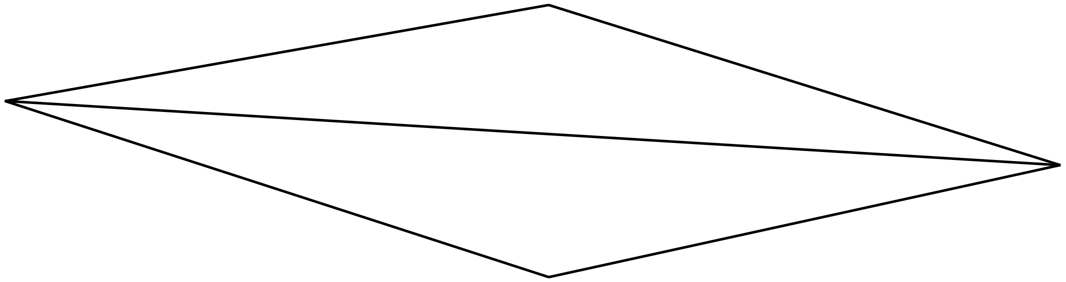
nonconvex polygon

Polygon Testing

- Conceptually simple to test for simplicity and convexity
- Time consuming
- Earlier versions assumed both and left testing to the application
- Present version only renders triangles
- Need algorithm to triangulate an arbitrary polygon

Good and Bad Triangles

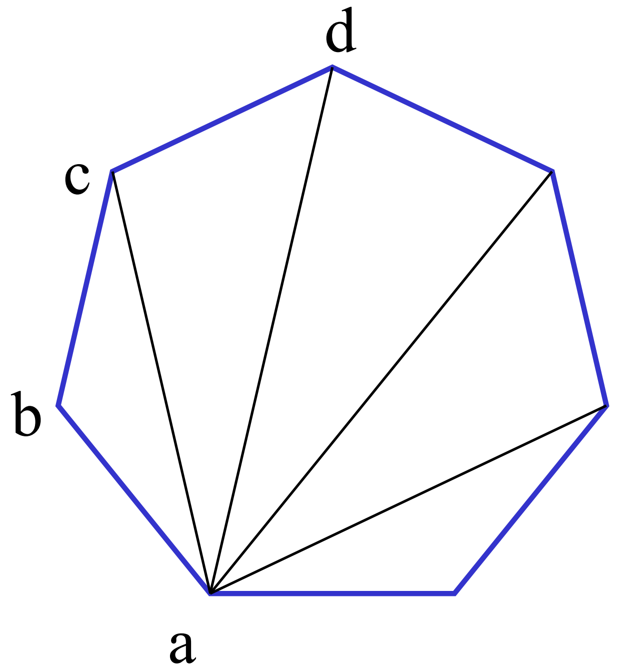
- Long thin triangles render badly



- Equilateral triangles render well
- Maximize minimum angle
- Delaunay triangulation for unstructured points

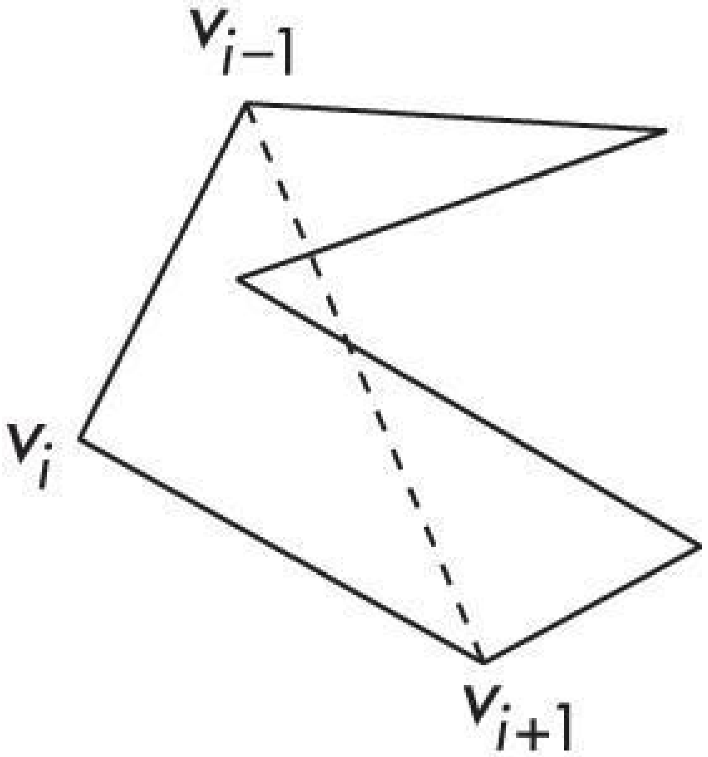
Triangularization

- Convex polygon



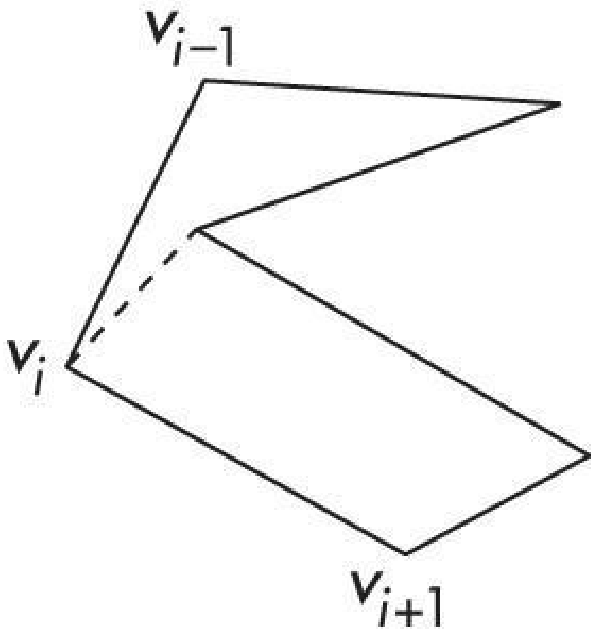
- Start with abc, remove b, then acd, …

Non-convex (concave)



Recursive Division

- Find leftmost vertex and split

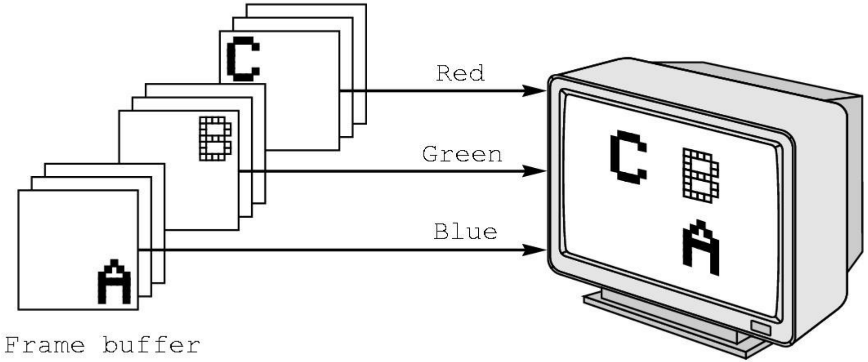


Attributes

- Attributes determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges
    - Display vertices
- Only a few (gl_PointSize) are supported by WebGL functions

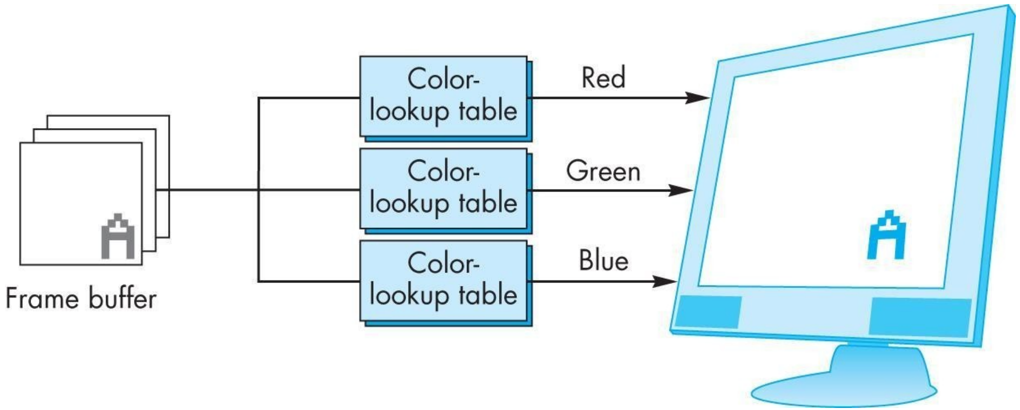
RGB color

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes



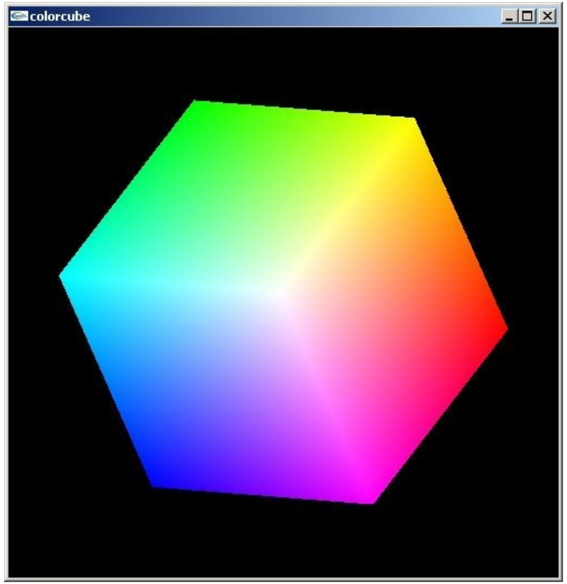
Indexed Color

- Colors are indices into tables of RGB values
- Requires less memory
    - -indices usually 8 bits
    - -not as important now
        - Memory inexpensive
        - Need more colors for shading



Smooth Color

- Default is smooth shading
  - Rasterizer interpolates vertex colors across visible polygons
- Alternative: flat shading
  - Color of first vertex determines fill color
  - Handle in shader



Setting Colors

- Colors are ultimately set in the fragment shader but can be determined in either shader or in the application
- Application color: pass to vertex shader as a uniform variable or as a vertex attribute
- Vertex shader color: pass to fragment shader as varying variable
- Fragment color: can alter via shader code

Programming with WebGL: More GLSL

Sorin Babii
sorin.babii@cs.upt.ro

Objectives

- Coupling shaders to applications
  - Reading
  - Compiling
  - Linking
- Vertex Attributes
- Setting up uniform variables
- Example applications

Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
    - Vertex attributes
    - Uniform variables

Program Object

- Container for shaders
  - -Can contain multiple shaders
  - -Other GLSL functions

```
var program = gl.createProgram();

gl.attachShader( program, vertShdr );
gl.attachShader( program, fragShdr );
gl.linkProgram( program );
```

Reading a Shader

- Shaders are added to the program object and compiled
- Usual method of passing a shader is as a null-terminated string using the function

```
gl.shaderSource( fragShdr, fragElem.text );
```

- If shader is in HTML file, we can get it into application by getElementById method
- If the shader is in a file, we can write a reader to convert the file to a string

Adding a Vertex Shader

```
var vertShdr;
var vertElem =
    document.getElementById( vertexShaderId );

vertShdr = gl.createShader( gl.VERTEX_SHADER );

gl.shaderSource( vertShdr, vertElem.text );
gl.compileShader( vertShdr );

// after program object created
gl.attachShader( program, vertShdr );
```

Precision Declaration

- In GLSL for WebGL we must specify desired precision in fragment shaders
  - artifact inherited from OpenGL ES
  - ES must run on very simple embedded devices that may not support 32-bit floating point
  - All implementations must support mediump
  - No default for float in fragment shader
- Can use preprocessor directives (`#ifdef`) to check if `highp` supported and, if not, default to mediump

Programming with WebGL: 3D

Sorin Babii
sorin.babii@cs.upt.ro

Objectives

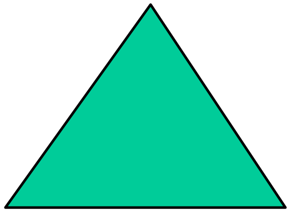
- Develop a more sophisticated 3D example
  - Sierpinski gasket: a fractal
- Introduce hidden-surface removal

3D Applications

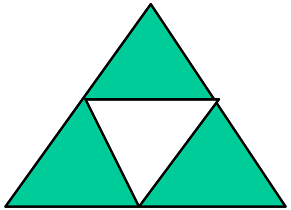
- In WebGL, 2D applications are a special case of 3D graphics
- Going to 3D
  - Not much changes
  - Use `vec3`, `gl.uniform3f`
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal

Sierpinski Gasket (2D)

- Start with a triangle



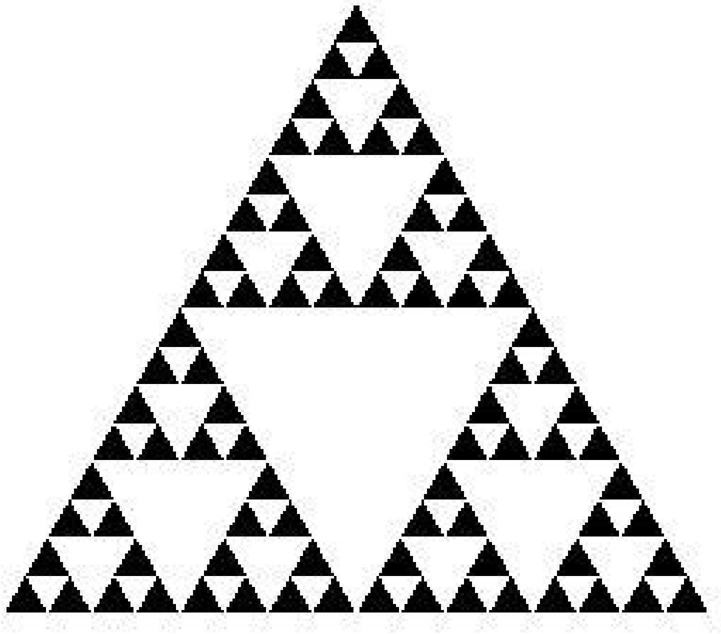
- Connect bisectors of sides and remove central triangle



- Repeat

Example

5 Steps



The gasket as a fractal

- Consider the filled area (black) and the perimeter (the length of all the lines around the filled triangles)
- As we continue subdividing
  - the area goes to zero
  - but the perimeter goes to infinity
- This is not an ordinary geometric object
  - It is neither two- nor three-dimensional
- It is a fractal (fractional dimension) object

Gasket Program

- HTML file - Same as in other examples
    - -Pass through vertex shader
    - -Fragment shader sets color
    - -Read in JS file

Gasket Program

```
var points = [];
var NumTimesToSubdivide = 5;

/* initial triangle */

 var vertices = [
       vec2( -1, -1 ),
       vec2(  0,  1 ),
       vec2(  1, -1 )
    ];

divideTriangle( vertices[0],vertices[1],
     vertices[2], NumTimesToSubdivide);
```

Draw one triangle

```
/* display one triangle  */

function triangle( a, b, c ){
    points.push( a, b, c );
}
```

Triangle Subdivision

```
function divideTriangle( a, b, c, count ){
 // check for end of recursion
    if ( count === 0 ) {
    triangle( a, b, c );
    }
    else {
//bisect the sides
    var ab = mix( a, b, 0.5 );
    var ac = mix( a, c, 0.5 );
    var bc = mix( b, c, 0.5 );

// three new triangles
    divideTriangle( a, ab, ac, count-1 );
    divideTriangle( c, ac, bc, count-1 );
    divideTriangle( b, bc, ab, count-1 );
    }
}
```

init()

```
var program = initShaders( gl, "vertex-shader",
                               "fragment-shader" );
    gl.useProgram( program );
var bufferId = gl.createBuffer();
    gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
    gl.bufferData( gl.ARRAY_BUFFER, flatten(points),
                   gl.STATIC_DRAW );

var vPosition = gl.getAttribLocation( program, "vPosition" );
   gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0 );
   gl.enableVertexAttribArray( vPosition );
   render();
```

Render Function

```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLES, 0, points.length );
}
```

Programming with WebGL: Moving to 3D

Sorin Babii
sorin.babii@cs.upt.ro

Moving to 3D

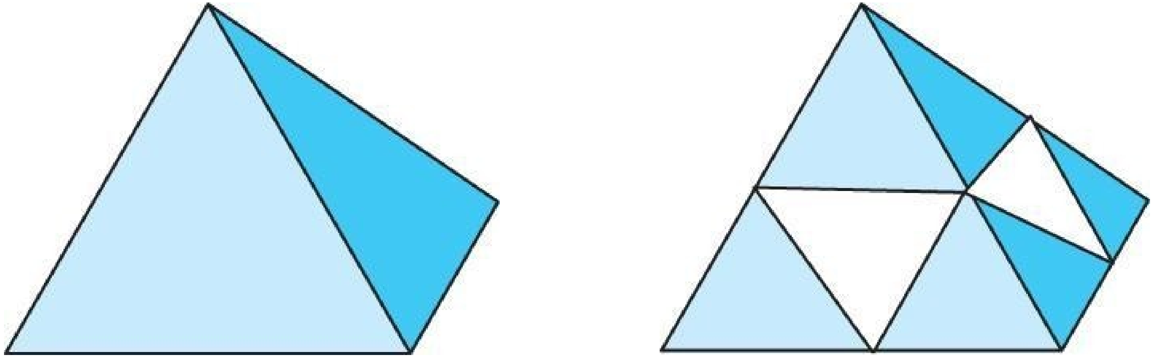
- We can easily make the program 3D by using three dimensional points and starting with a tetrahedron

```
var vertices = [
  vec3(  0.0000,  0.0000, -1.0000 ),
  vec3(  0.0000,  0.9428,  0.3333 ),
  vec3( -0.8165, -0.4714,  0.3333 ),
  vec3(  0.8165, -0.4714,  0.3333 )
];
```

subdivide each face

3D Gasket

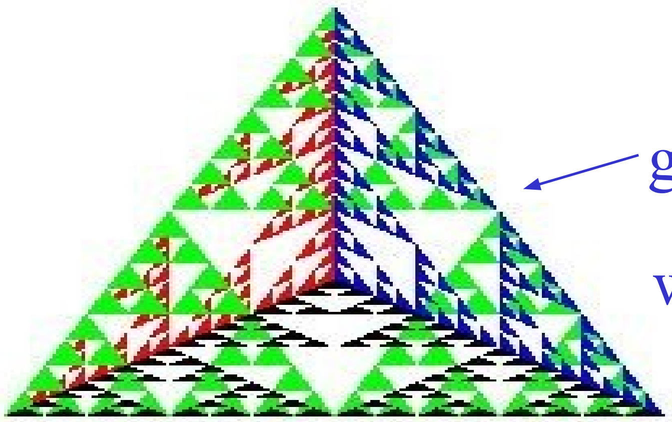
- We can subdivide each of the four faces

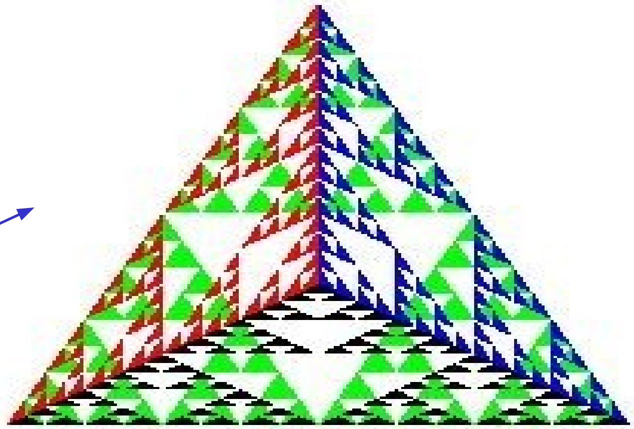


- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra
- Code almost identical to 2D example

Almost Correct

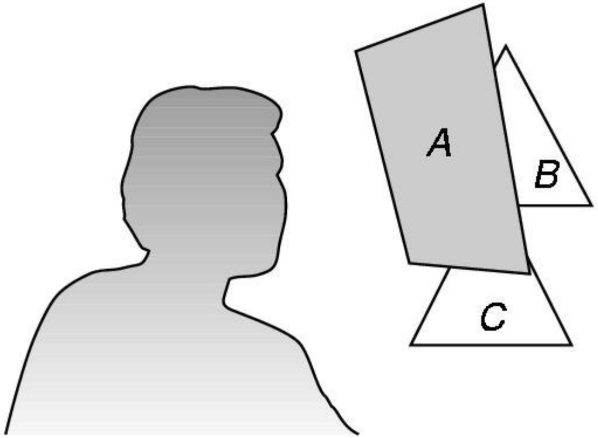
- Because the triangles are drawn in the order they are specified in the program, the front triangles are not always rendered in front of triangles behind them





Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces
- OpenGL uses a hidden-surface method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image



Using the z-buffer algorithm

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- Depth buffer is required to be available in WebGL
- It must be
  - Enabled
    - gl.enable(gl.DEPTH_TEST)
  - Cleared in for each render
    - gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)

Surface vs Volume Subdvision

- In our example, we divided the surface of each face
- We could also divide the volume using the same midpoints
- The midpoints define four smaller tetrahedrons, one for each vertex
- Keeping only these tetrahedrons removes a volume in the middle

Volume Subdivision