Chapter 2 Graphics Programming

Programming with OpenGL or WebGL

Key Content

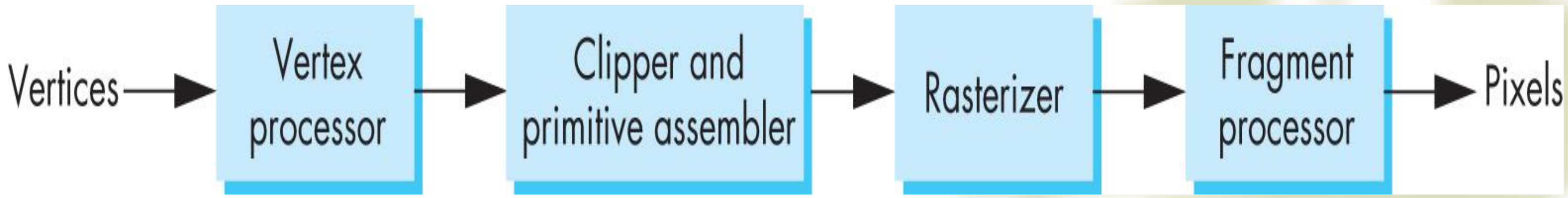
1. OpenGL Evolution
2. OpenGL Libraries
3. OpenGL Introduction
4. OpenGL Primitives
5. OpenGL Shaders Programming
6. 2D Sierpinski Gasket
7. Application

OpenGL Evolution

- Originally controlled by an Architectural Review Board (ARB)
  - Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM,…….
  - Now Kronos Group
  - Was relatively stable (through version 2.5)
    - Backward compatible
    - Evolution reflected new hardware capabilities
      - 3D texture mapping and texture objects
      - Vertex and fragment programs

Modern OpenGL

- Performance is achieved by using GPU rather than CPU
- Control GPU through programs called shaders
- Application's job is to send data to GPU
- GPU does all rendering



OpenGL 3.1

- Totally shader-based
  - No default shaders
  - Each application must provide both a vertex and a fragment shader
- No immediate mode
- Few state variables
- Most 2.5 functions deprecated
- Backward compatibility not required

Other Versions

- OpenGL ES
  - Embedded systems
  - Version 1.0 simplified OpenGL 2.1
  - Version 2.0 simplified OpenGL 3.1
    - Shader based
- WebGL
  - Javascript implementation of ES 2.0
  - Supported on newer browsers
- OpenGL 4.1 and 4.5, 4.6 in Oct. 2019
  - Add geometry shaders and tessellator

What About Direct X?

- Windows only
- Advantages
  - Better control of resources
  - Access to high level functionality
- Disadvantages
  - New versions not backward compatible
  - Windows only
- Recent advances in shaders are leading to convergence with OpenGL

2. OpenGL Libraries

- OpenGL core library
  - OpenGL32 on Windows
  - GL on most unix/linux systems (libGL.a)
- OpenGL Utility Library (GLU)
  - Provides functionality in OpenGL core but avoids having to rewrite code
  - Will only work with legacy code
- Links with window system
  - GLX for X window systems
  - WGL for Windows
  - AGL for Macintosh

GLUT

- OpenGL Utility Toolkit (GLUT)
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
    - Event-driven
  - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
    - No slide bars

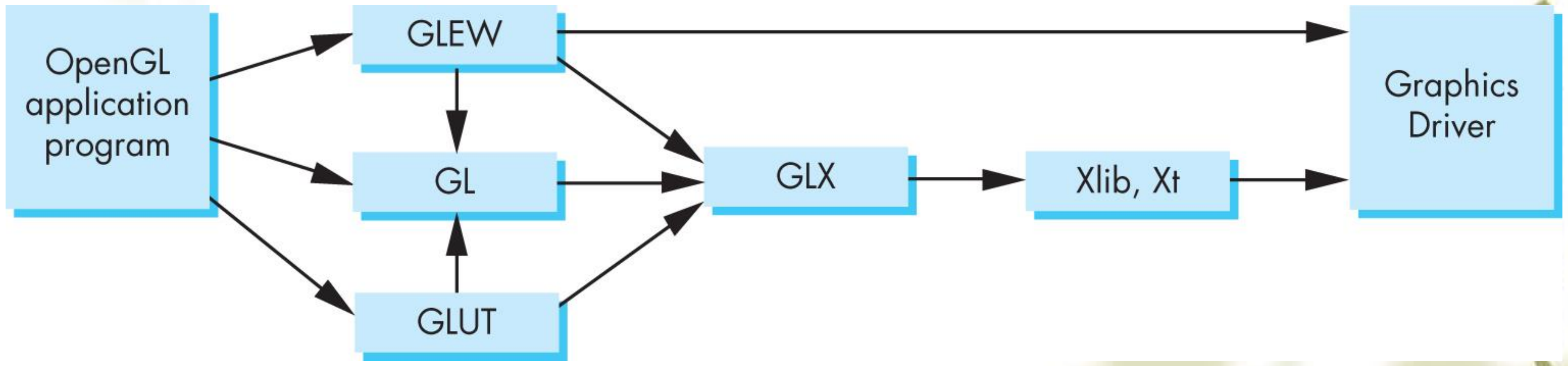
freeglut

- GLUT was created long ago and has been unchanged
  - Amazing that it works with OpenGL 3.1
  - Some functionality can’t work since it requires deprecated functions
- **freeglut** updates GLUT
  - Added capabilities
  - Context checking

GLEW

- OpenGL Extension Wrangler Library
- Makes it easy to access OpenGL extensions available on a particular system
- Avoids having to have specific entry points in Windows code
- Application needs only to include glew.h and run a glewInit()

Libraries Organization



WebGL

- WebGL is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES, exposed to ECMAScript via the HTML5 Canvas element
- WebGL 2.0 exposes the OpenGL ES 3.0 API
- See also appendix A for GLSL with WebGL

https://www.khronos.org/webgl/

3. OpenGL Introduction

- Primitives
  - Points
  - Line Segments
  - Triangles
- Attributes
- Transformations
  - Viewing
  - Modeling
- Control (freeglut)
- Input (freeglut)
- Query

OpenGL State

- OpenGL is a state machine
- OpenGL functions are of two types
  - Primitive generating
    - Can cause output if primitive is visible
    - How vertices are processed and appearance of primitive are controlled by the state
  - State changing
    - Transformation functions
    - Attribute functions
    - Under 3.1 most state variables are defined by the application and sent to the shaders

OpenGL State

- OpenGL is a state machine
- OpenGL functions are of two types
  - Primitive generating
    - Can cause output if primitive is visible
    - How vertices are processed and appearance of primitive are controlled by the state
  - State changing
    - Transformation functions
    - Attribute functions

Examples for changing states of OpenGL:

Setting State

```
        glPointSize( size );
        glLineStipple( repeat, pattern );
        glShadeModel( GL_SMOOTH );
```

Enabling Features

```
        glEnable( GL_LIGHTING );
        glDisable( GL_TEXTURE_2D );
```

Lack of Object Orientation

- OpenGL is not object oriented so that there are multiple functions for a given logical function
  - **`glUniform3f`**
  - **`glUniform2i`**
  - **`glUniform3dv`**
- Underlying storage mode is the same
- Easy to create overloaded functions in C++ but issue is efficiency

OpenGL function format

function name

dimensions

glUniform3f(x,y,z)

belongs to GL library

x,y,z are floats

glUniform3fv(p)

p is a pointer to **an array**

Grammar of OpenGL Functions

Lib name + Command name

+{dimension: 2,3,4}

+{parameter data type: sifdb(wb)}

+{array identification: v} (parameter)

Lib name: gl, glu, freeglut, glew

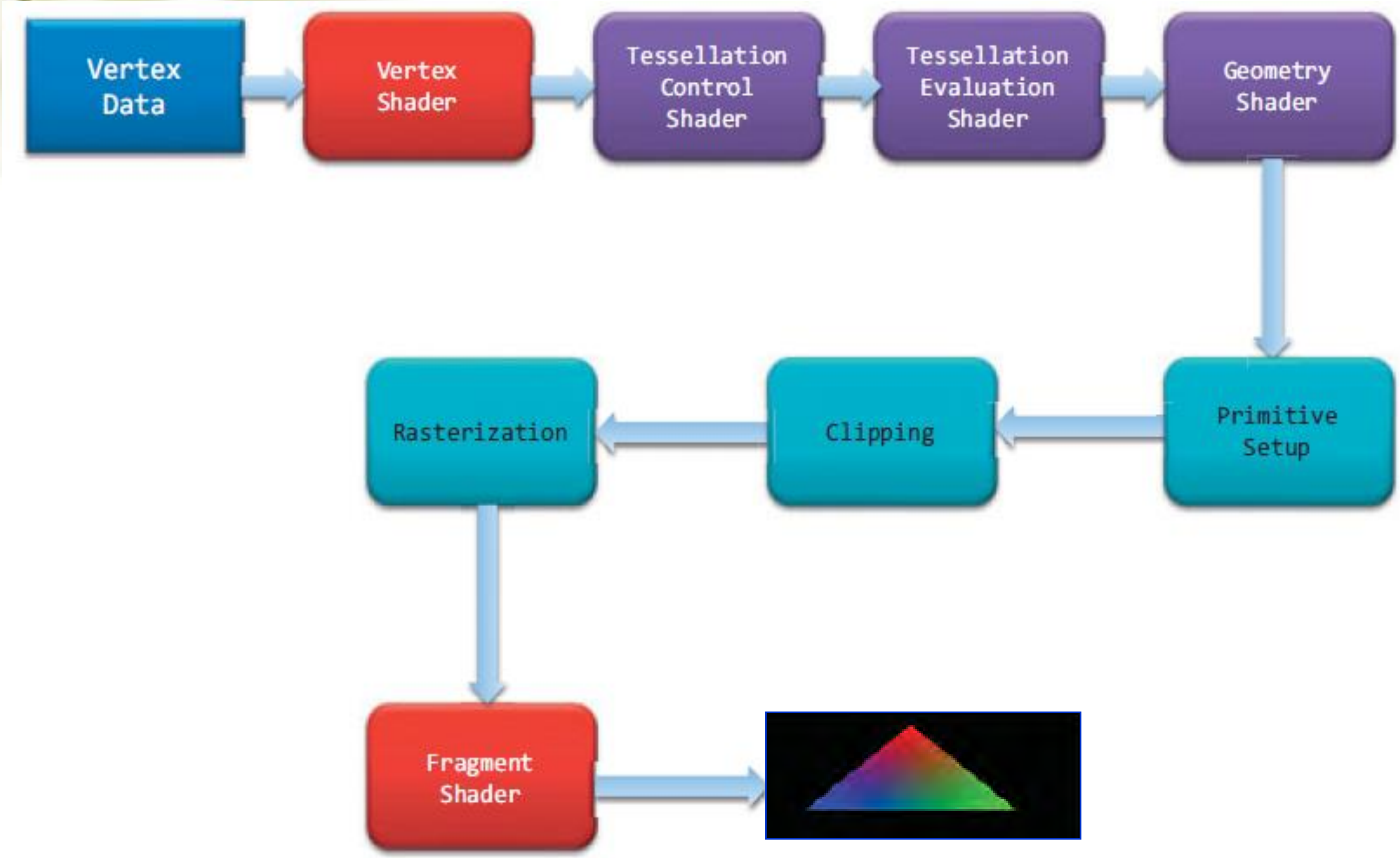
Ex. glVertex3f(5.,6.,7.);

  glBindVertexArray(abuffer)

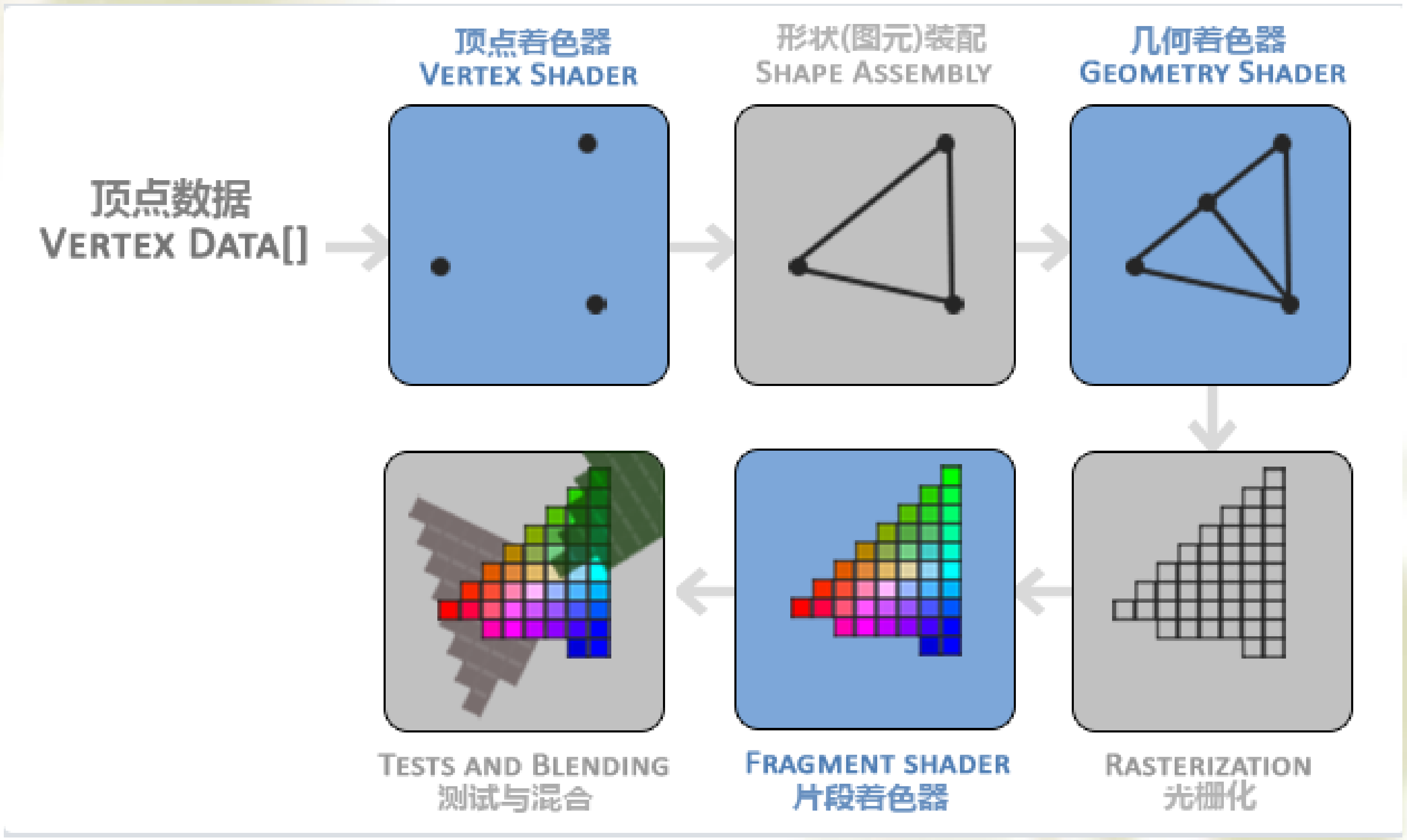
GLSL

- Open<u>GL</u> <u>S</u>hading <u>L</u>anguage
- C-like with
  - Matrix and vector types (2, 3, 4 dimensional)
  - Overloaded operators
  - C++ like constructors
- Similar to Nvidia's **Cg** and Microsoft **HLSL**
- Code sent to shaders as source code
- New OpenGL functions to compile, link and get information to shaders

OpenGL and GLSL



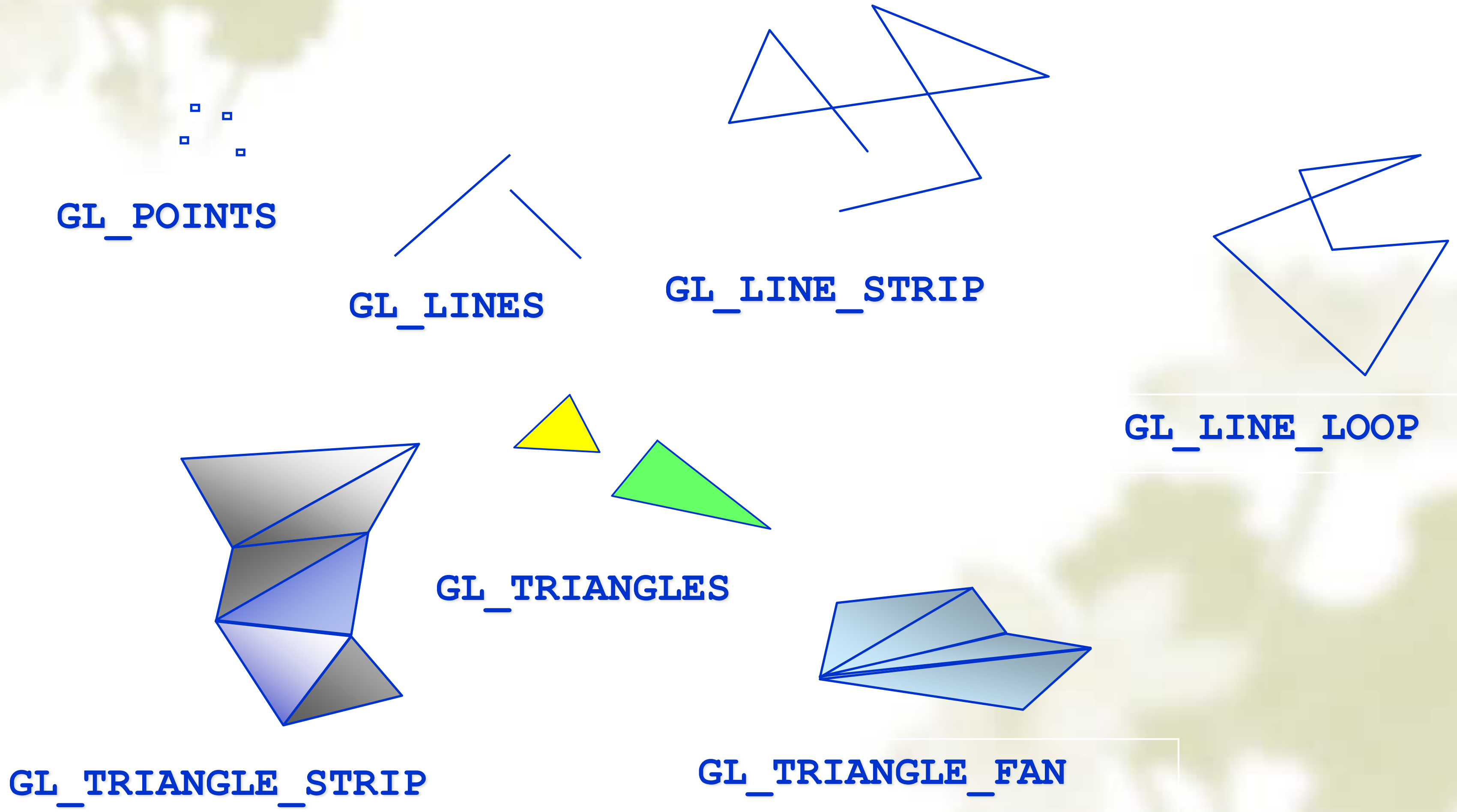
OpenGL and GLSL



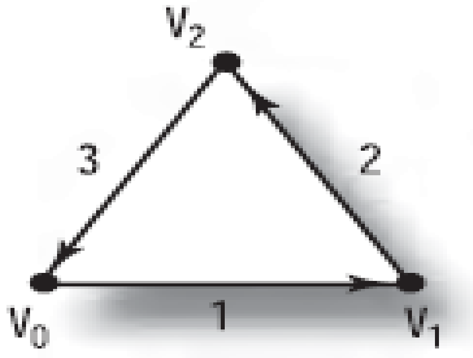
OpenGL Resources

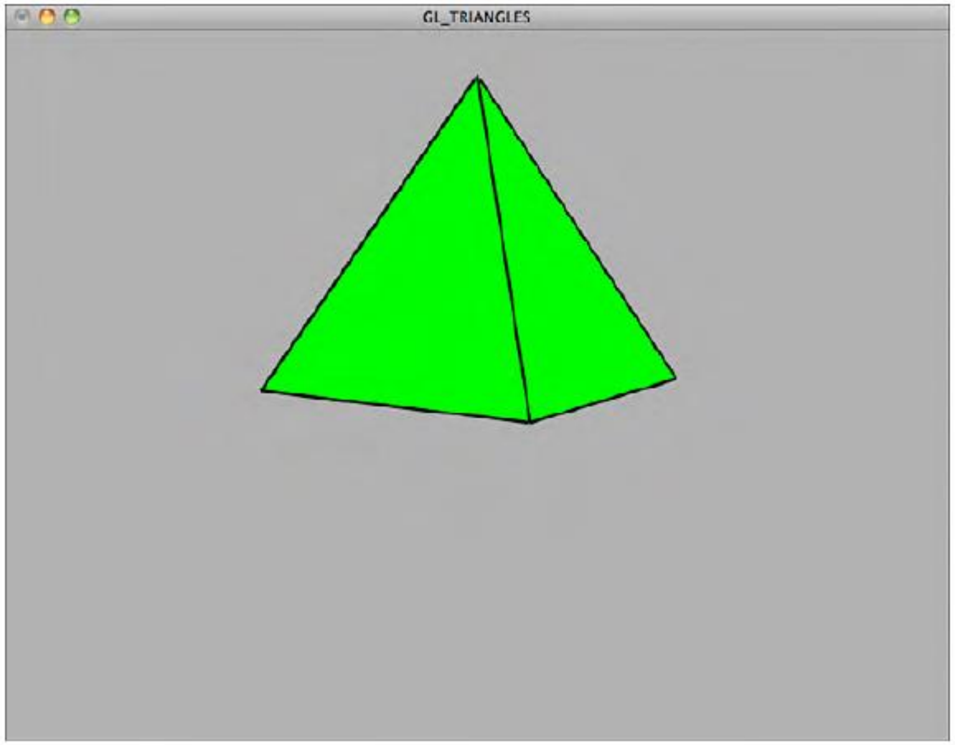
- Website: *http://www.opengl.org/*
- Reference book “OpenGL Programming Guide 8th Edition”
- Reference book “OpenGL SuperBible 7th Edition”

4. OpenGL Primitives

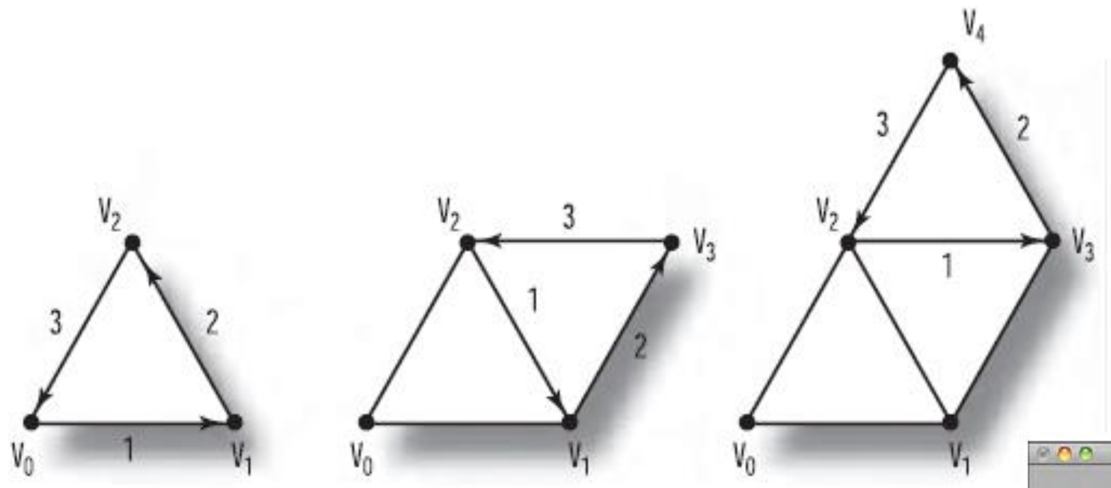


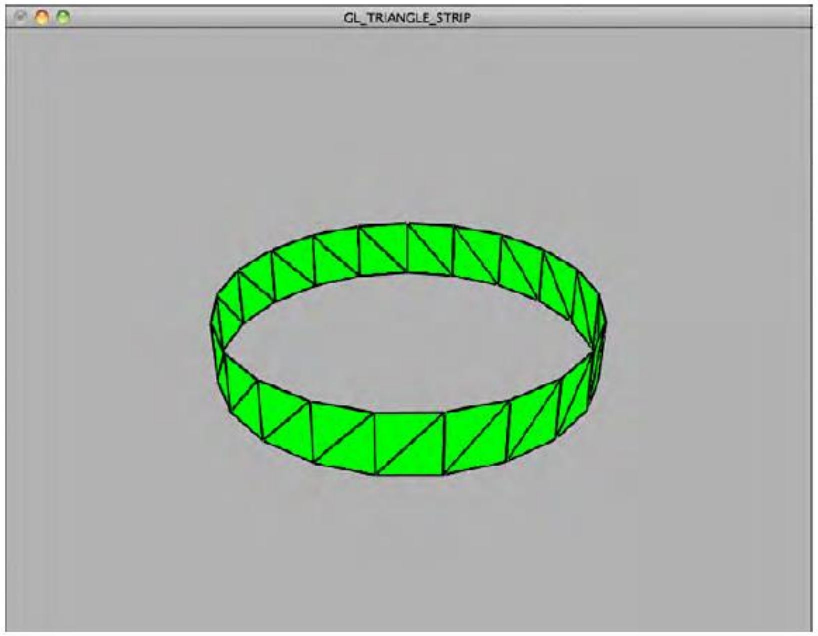
GL_TRIANGLES



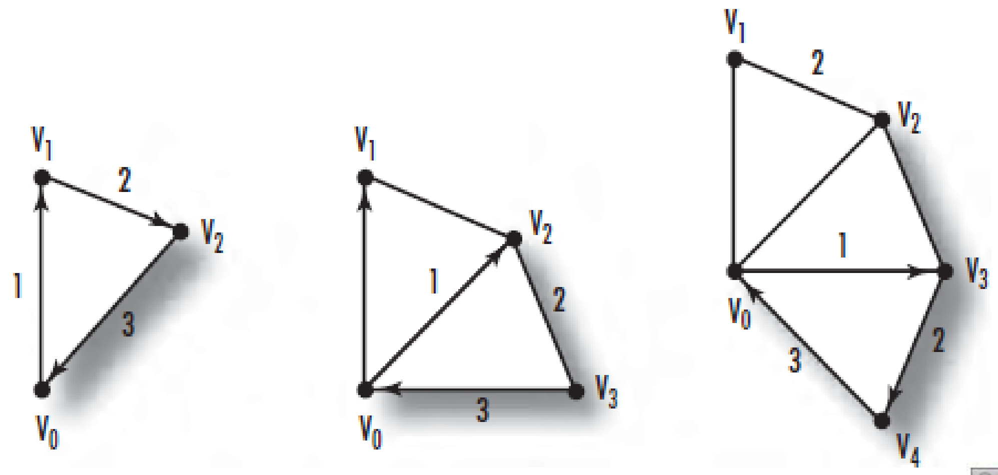


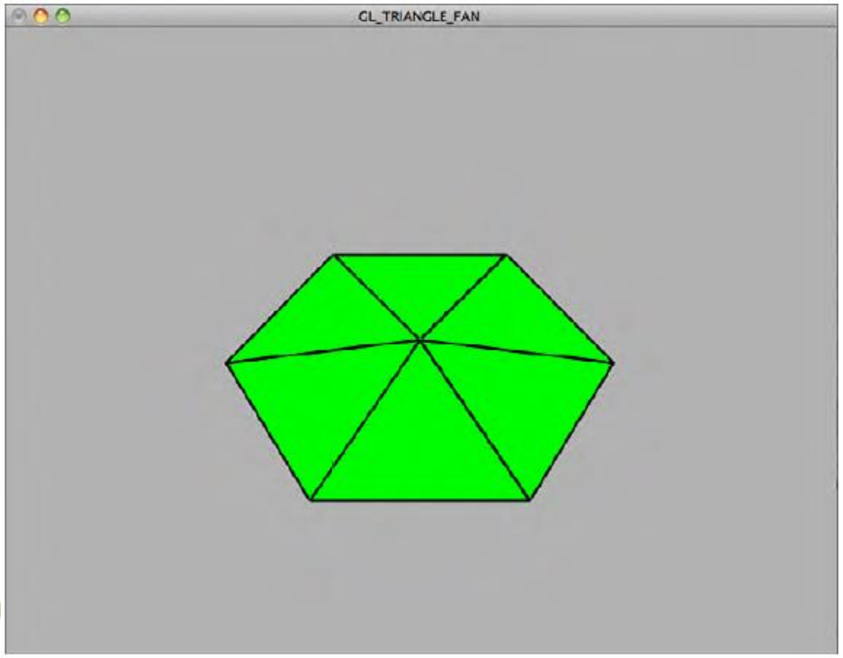
GL_TRIANGLE_STRIP





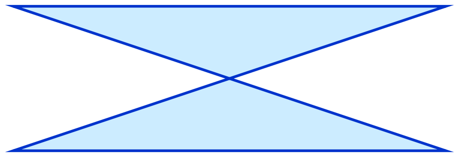
GL_TRIANGLE_FAN



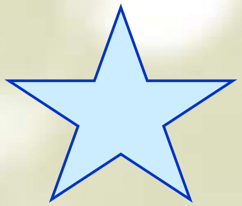


Polygon Issues

- OpenGL 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



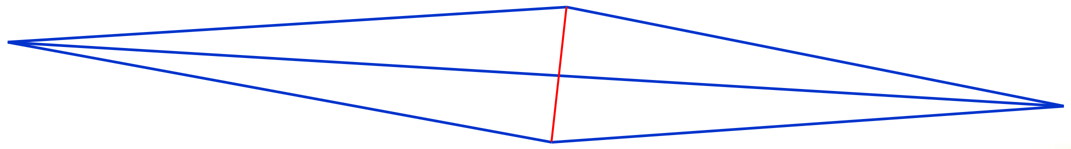
nonsimple polygon



nonconvex polygon

Good and Bad Triangles

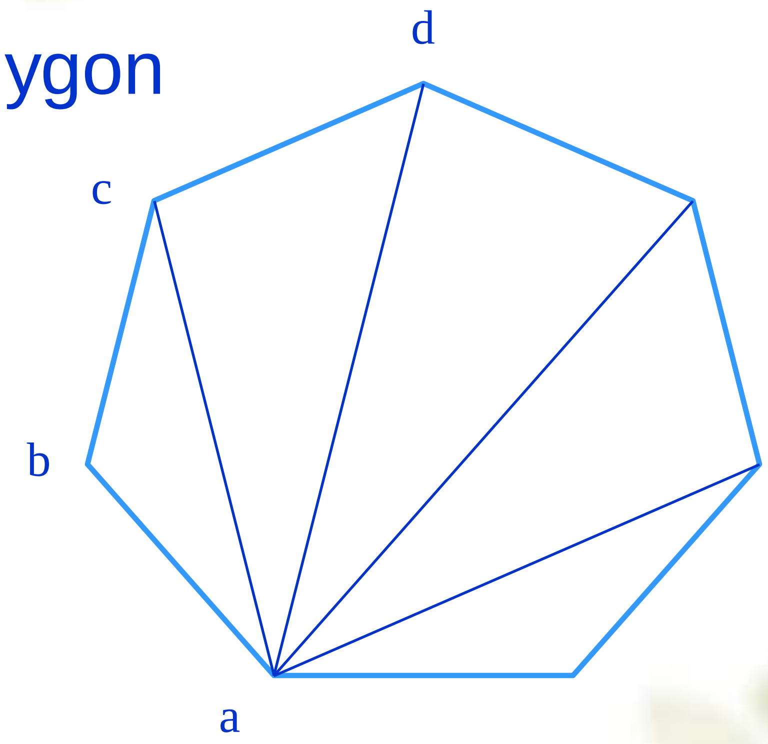
- Long thin triangles render badly



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

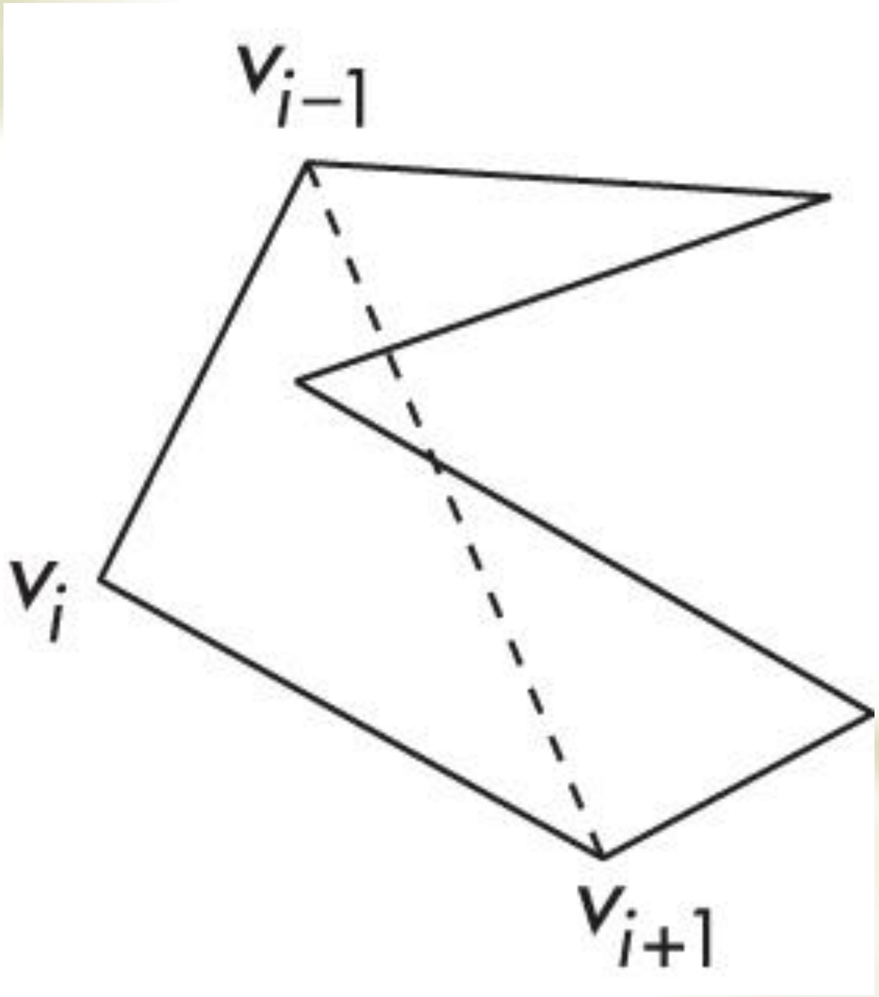
Triangularization

- Convex polygon



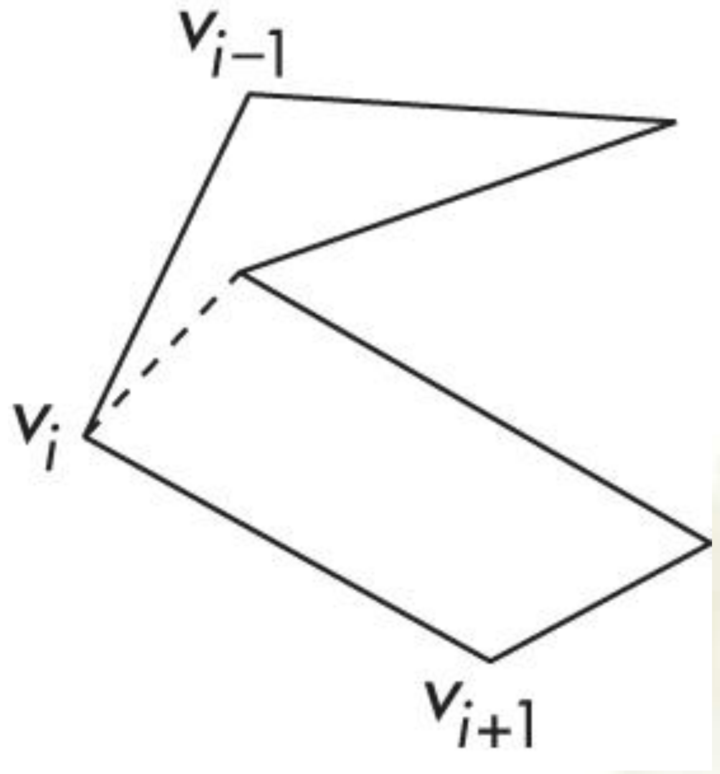
- Start with abc, remove b, then acd, ….

Non-convex (concave)



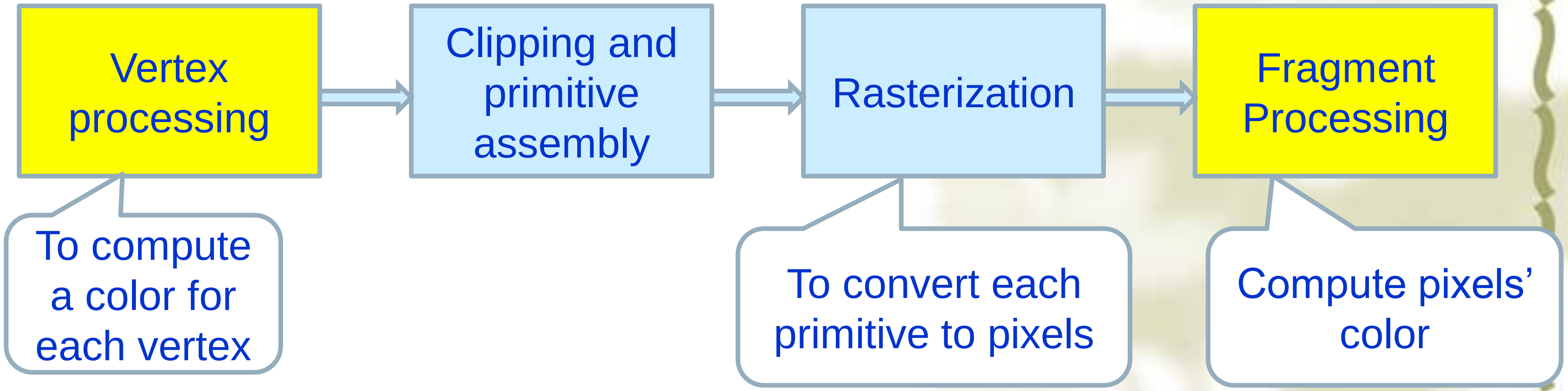
Recursive Division

- Find leftmost vertex and split



5. OpenGL Shaders Programming

- Present version only renders triangles
- Better to send **array** over and store on GPU with **buffer** for multiple renderings


Vertex processing
Clipping and primitive assembly
Rasterization
Fragment Processing
To compute a color for each vertex
To convert each primitive to pixels
Compute pixels' color

Program Structure—first part

```
#include <iostream>
using namespace std

#include "LoadShaders.h"

enum VAO_IDs {Triangle, NumVAOs};
enum Buffer_IDs { ArrayBuffer, NumBuffers };
enum Attrib_IDs {vPosition =0};

GLuint  VAOs[NumVAOs];
GLuint  Buffers[NumBuffers];

Const Gluint NumVertices = 3;
```

Vertex Array Object

Buffer in Shader

What attributes in Shader

```
// init

void  init (void)
{
   glGenVertexArrays(NumVAOs, VAOs);
   glBindVertexArray(VAOs[Triangle]);

   Glfloat  vertices[NumVertices][2] = { { -0.7, -0.7}, { 0.0, 0.7 }, { 0.7, -0.7}};

   glGenBuffers(NumBuffers, Buffers);
   glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);
   glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),vertices);

   ShaderInfo  shaders[] = {  {GL_VERTEX_SHADER, “triangle.vert” },
                              { GL_FRAGMENT_SHADER, “triangle.frag”},
                              {GL_NONE, NULL}  };
   GLuint  program = LoadShaders(shaders);
   glUseProgram(program);

   glVertexAttribPointer(vPosition, 0, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
   glEnableVertexAttribArray(vPosition);
}
```

Object names

Buffer names

Data=>Buffer

Creating shader program

Buffer related to vPosition in shader

```
//  mydisplay

Void  mydispaly(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glBindVertexArray(VAOs[Triangle]);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);

    glFlush();
}
```

Type of primitives

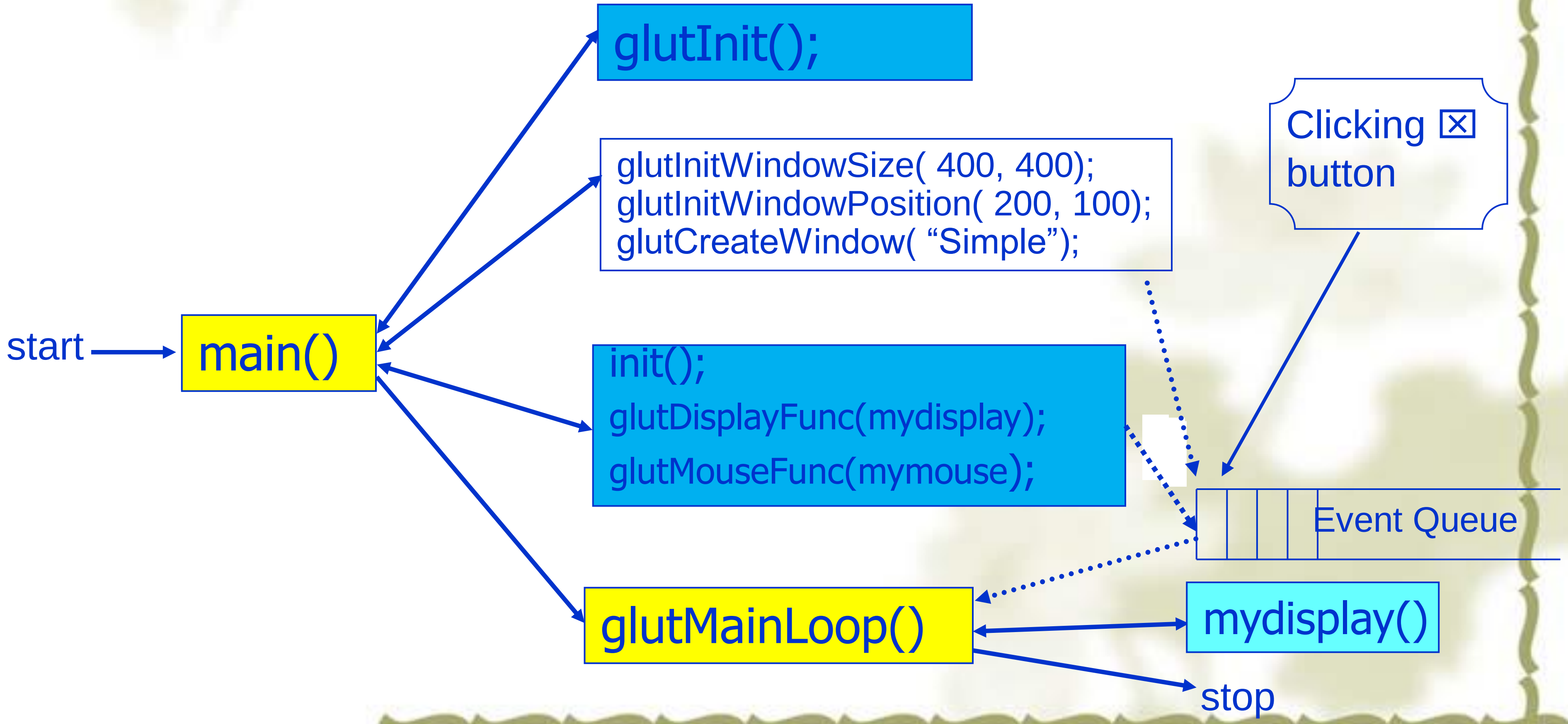
Draw in shaders

```
//  main

int  main(int  argc,  char **  argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA);
    glutInitWindowSize (500,500);
    glutInitContextVersion(4,3);
    glutInitContextProfile(GLUT_CORE_PROFILE);
    glutCreateWindow(“Simple”);

    if(glewInit())  {  cerr<<“error glew” << endl;
            exit(EXIT_FAILURE);
    }

   init();

   glDisplayFunc(mydisplay);

   glMainLoop();
}
```

Callback function to process events

Execution sequence

At first, glutMainLoop() picks up a <**window creation**> event from the queue, creates the window, and calls mydisplay(). When the ☒ button of the window is clicked, a <**stop**> event is inserted in the queue. When this event is processed, the execution terminates



```
glutInit();
```

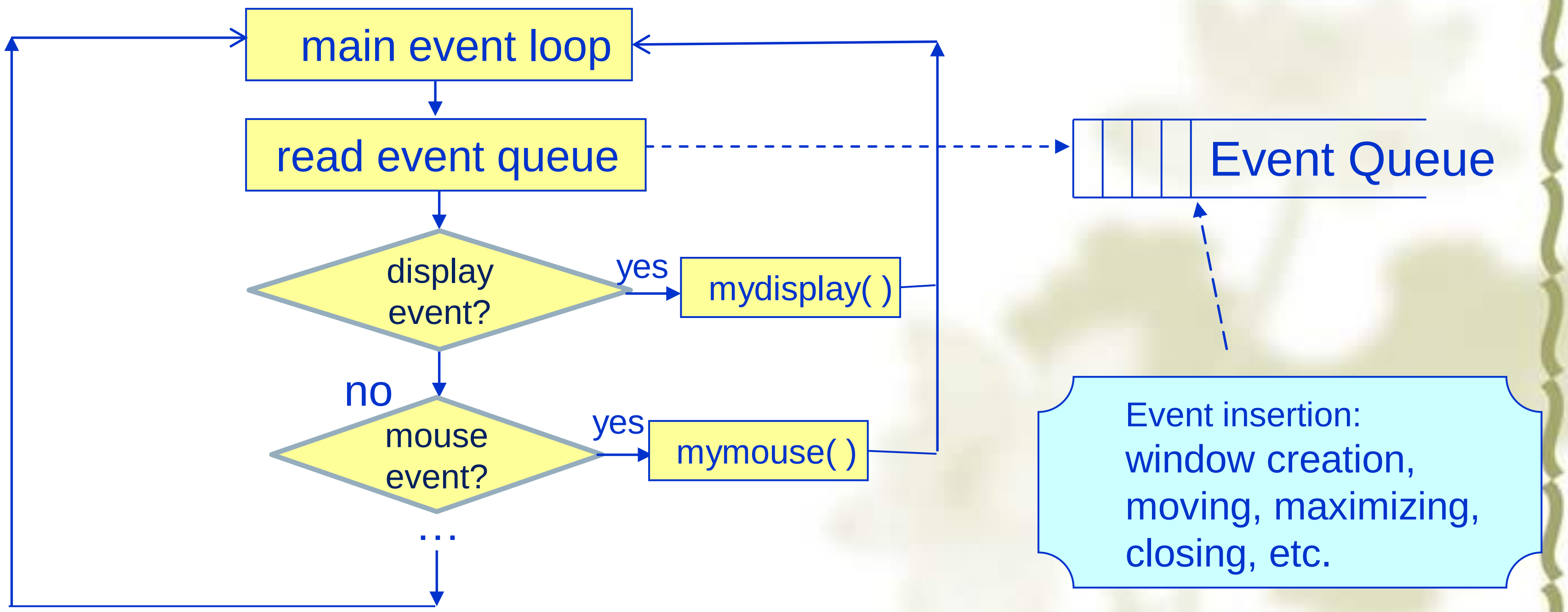
```
glutInitWindowSize( 400, 400);
glutInitWindowPosition( 200, 100);
glutCreateWindow( “Simple”);
```

```
init();
glutDisplayFunc(mydisplay);
glutMouseFunc(mymouse);
```

```
glutMainLoop()
```

Event Processing

The function *glutMainLoop()* iterates indefinitely. In each iteration, it check whether there are any events in the queue. If yes, it removes and processes the first event. It terminates the execution of the program when a <stop> event is processed.



Event Loop

- the program specifies a *display callback* function named **`mydisplay-`** *glutDisplayFunc(**mydisplay**);*
  - Every glut program must have a display callback
  - The display callback is executed whenever OpenGL decides the display must be refreshed, for example when the window is opened
  - The **`main`** function ends with the program entering an event loop-- *glutMainLoop();*

Display Callback

- Once we get data to GPU, we can initiate the rendering with a simple callback

```
void mydisplay()
{
        glClear(GL_COLOR_BUFFER_BIT);

        glBindVertexArray(VAOs[Triangle]);
        glDrawArrays(GL_TRIANGLES, 0, NumVertices);

        glFlush();
}
```

- Arrays are buffer objects that contain vertex arrays

Vertex Arrays

- Vertices can have many attributes
  - Position
  - Color
  - Texture Coordinates
  - normal
- A vertex array holds these data

Vertex Array Object(VAO)

- Bundles all vertex data (positions, colors, ..,)
- Get name for buffer then bind

```
glGenVertexArrays(NumVAOs, VAOs);
glBindVertexArray(VAOs[Triangle]);
```

- At this point we have a current vertex array but no contents
- Use of glBindVertexArray lets us switch between VBOs（ Vertex Buffer Object – in GPU）

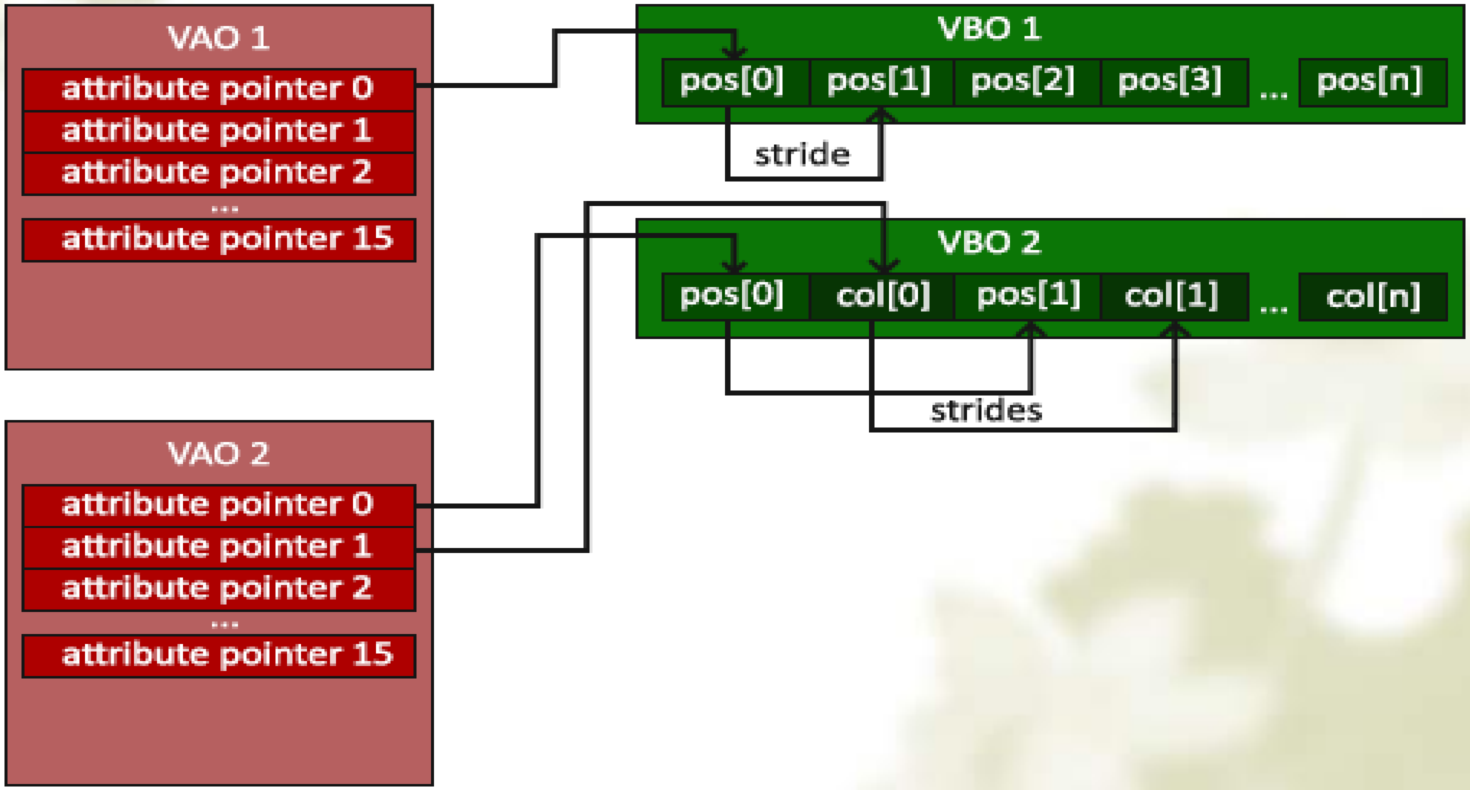
Vertex Buffer Object(VBO)

- Buffers objects allow us to transfer large amounts of data to the GPU
- Need to create, bind and identify data

```
glGenBuffers(NumBuffers, Buffers);
glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),vertices);
```

- Data in current vertex array is sent to GPU

VAO and VBO


VAO 1
attribute pointer 0
attribute pointer 1
attribute pointer 2
...
attribute pointer 15
VAO 2
attribute pointer 0
attribute pointer 1
attribute pointer 2
...
attribute pointer 15
VBO 1
pos[0]
pos[1]
pos[2]
pos[3]
...
pos[n]
stride
VBO 2
pos[0]
col[0]
pos[1]
col[1]
...
col[n]
strides

Vertex Shader—second part

input from application

```
in vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;
}
```

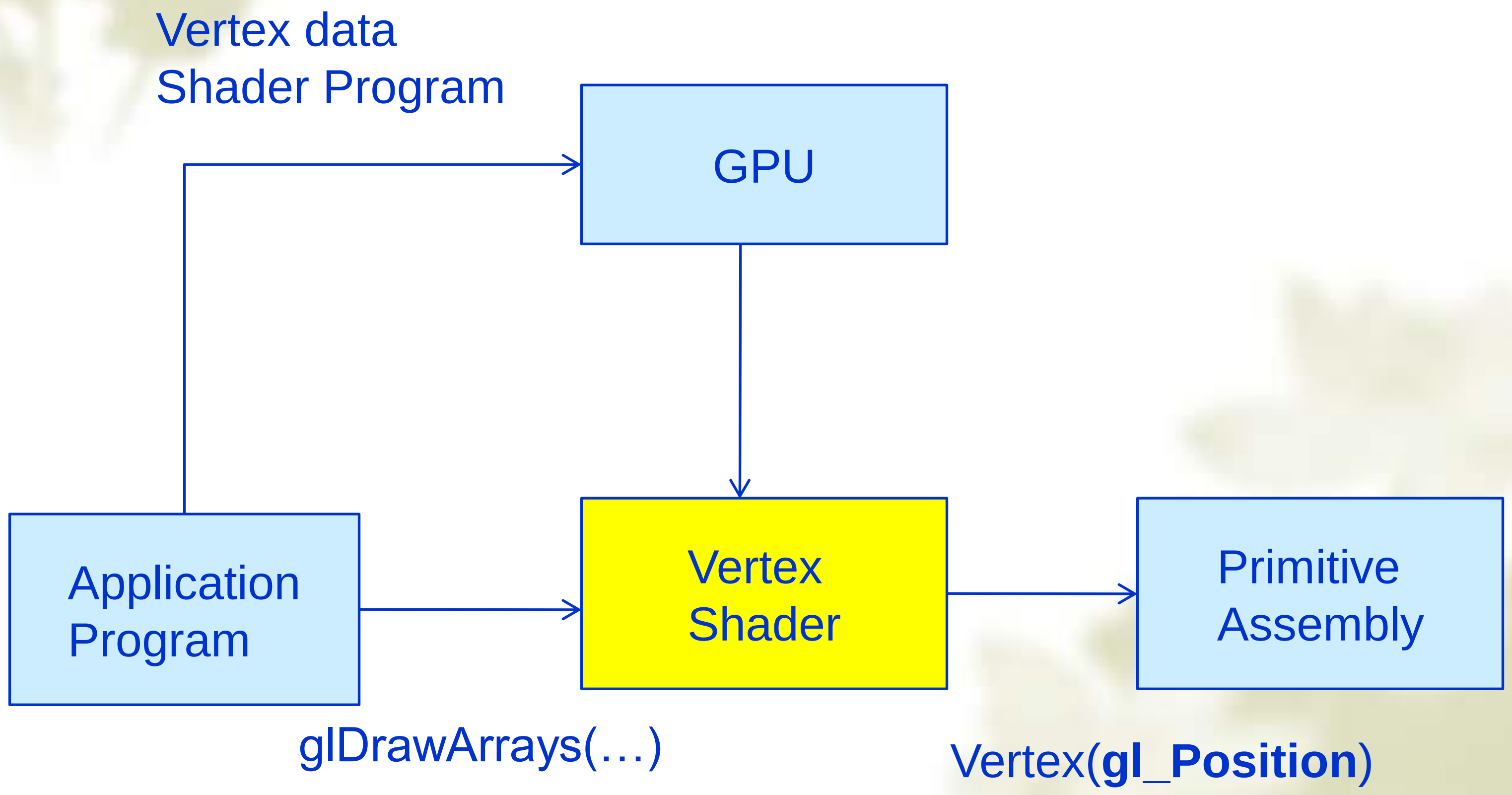
must link to variable in application

programmable

state variable in vertex shader, be output from the vertex shader

gl_Position is defined by OpenGL, which don't need to declare in shader

Execution Model

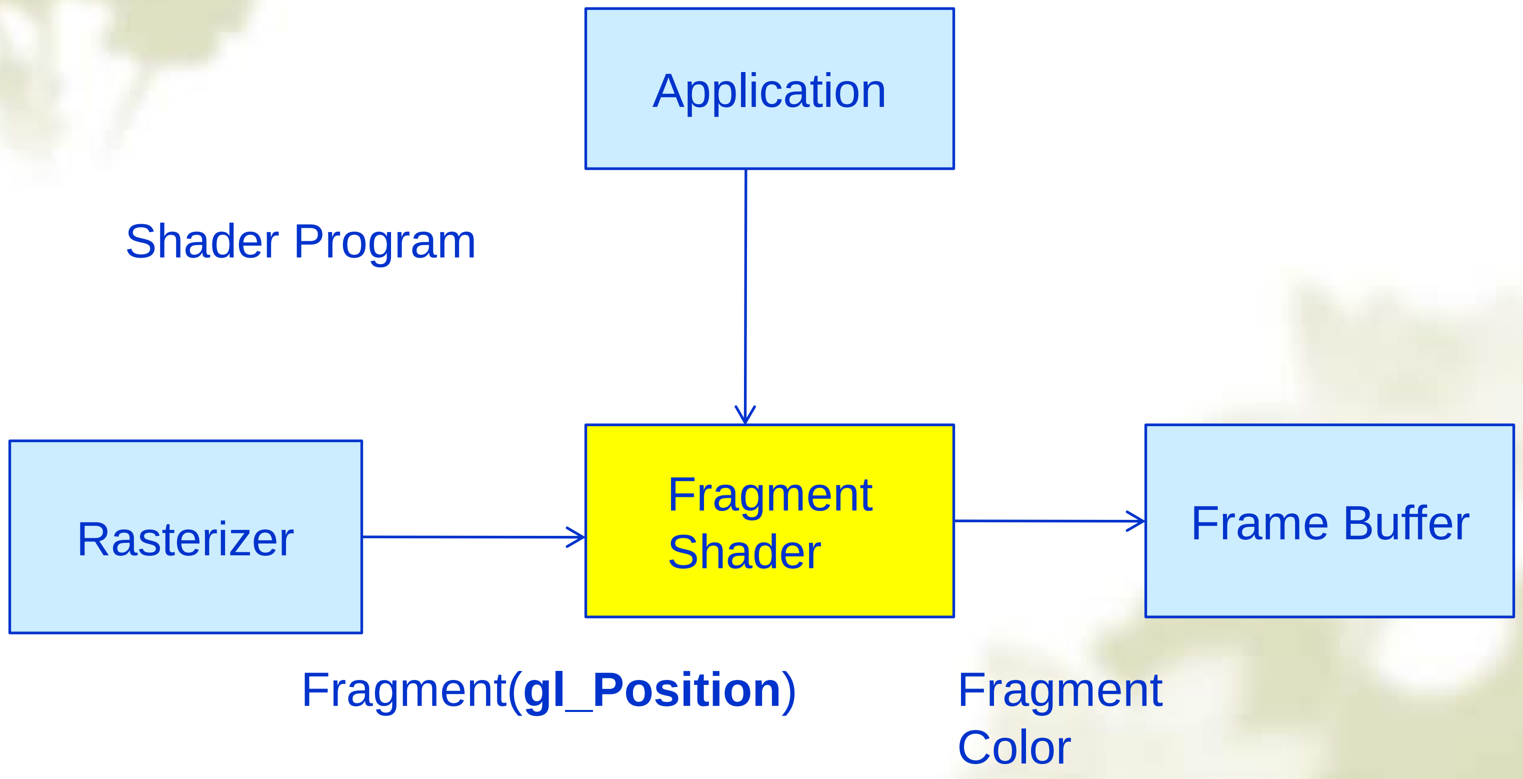


Fragment Program—third part

```
void main(void)
{
  gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

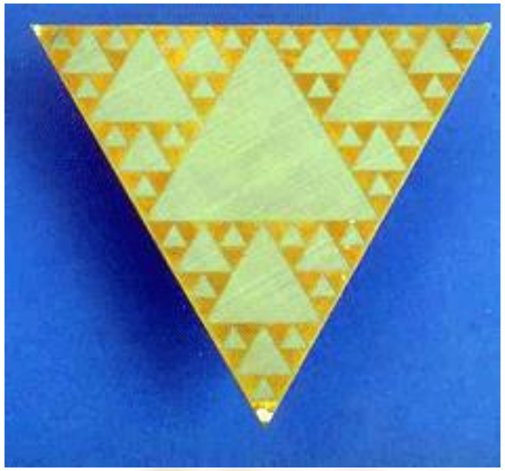
//set vPosition as green

Execution Model

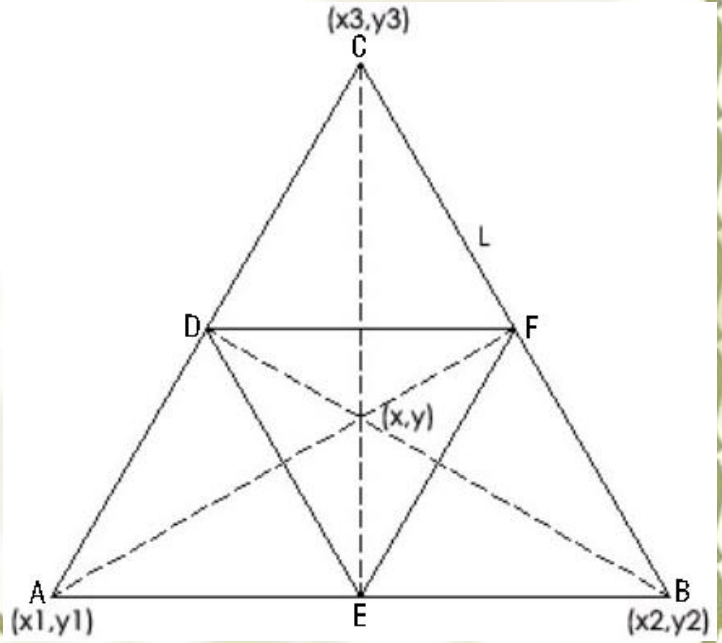


6. 2D Sierpinski gasket

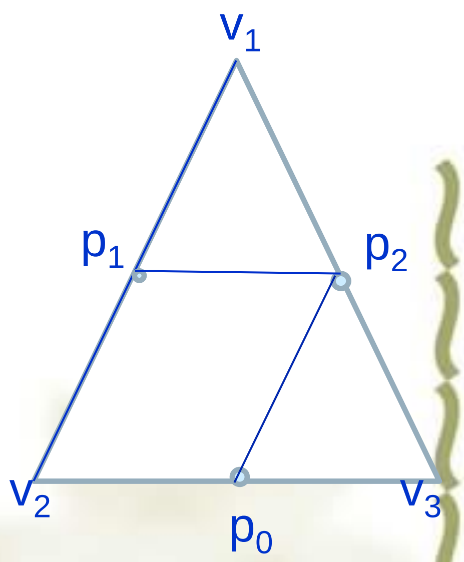
```
for(all_triangles)
{
      if(Not_End)  {
          st=generate_triangle(t);
          display(st);
          t = st;
      }
}
```



```
for(all_triangles)
{
      if(Not_End)  {
          st=generate_triangle(t);
          store_triangles(st);
          t = st;
      }
}
send_all_triangles_toGPU();
display_all();
```



```
// Two-Dimensional Sierpinski Gasket  see also 2.1
// Generated using randomly selected vertices and bisection

#include "Angel.h"

const int NumPoints = 5000;

void  init( void )
{
    vec2 points[NumPoints];

    // Specifiy the vertices for a triangle
    vec2 vertices[3] = {
            vec2( -1.0, -1.0 ), vec2( 0.0, 1.0 ), vec2( 1.0, -1.0 )
    };

    // Select an arbitrary initial point inside of the triangle
    points[0] = vec2( 0.25, 0.50 );

    // compute and store N-1 new points
    for ( int i = 1; i < NumPoints; ++i ) {
        int  j = rand() % 3;   // pick a vertex at random

        // Compute the point halfway between the selected vertex and the previous point
        points[i] = ( points[i - 1] + vertices[j] ) / 2.0;
    }
```



Continue in init()

```
    // Create a vertex array object(VAO)
    GLuint  vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    // Create and initialize a buffer object
    GLuint  buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );

    // Load shaders and use the resulting shader program
    GLuint program = InitShader( "vshader21.glsl", "fshader21.glsl" );
    glUseProgram( program );

    // Initialize the vertex position attribute from the vertex shader
    GLuint loc = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( loc );
    glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );

    glClearColor( 1.0, 1.0, 1.0, 1.0 ); // white background
}
```

```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );              // clear the window

    glDrawArrays( GL_POINTS, 0, NumPoints );    // draw the points

    glFlush();
}


void
keyboard( unsigned char key, int x, int y )
{
    switch ( key ) {
    case 033:
        exit( EXIT_SUCCESS );
        break;
    }
}
```

```
int  main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGBA );
    glutInitWindowSize( 512, 512 );

    // If you are using freeglut, the next two lines will check if
    // the code is truly 3.2. Otherwise, comment them out
    glutInitContextVersion( 3, 2 );
    glutInitContextProfile( GLUT_CORE_PROFILE );

    glutCreateWindow( "Sierpinski Gasket" );

    glewInit();

    init();

    glutDisplayFunc( display );
    glutKeyboardFunc( keyboard );

    glutMainLoop();

    return 0;
}
```

Vertex shader

```
#version 150

in vec4 vPosition;

void  main()
{
    gl_Position = vPosition;
}
```

Fragment shader

```
#version 150

out vec4  fColor;

void  main()
{
   fColor = vec4( 0.0, 1.0, 0.0, 1.0 );
}
```

作业2

1. 第1章作业第1.8题

2. Installation OpenGL Lib in visual studio 2010+, then writing a green triangle program with OpenGL Shader or WebGL with JavaScript three.js.

3. Program the 3D Sierpinski Gasket—see §2.8, §2.9 for 2D