Graphics Hardware and Shaders
“We’re very worried that John’s homework has started to interfere with his computer gaming.”

“I liked it better when he was a couch potato gamer. It was easier on the rug.”
Computer graphics (circa 1951)

http://design.osu.edu/carlson/history/tree/images/pages/whirlwind_jpeg.htm
Computer graphics (1956)

SAGE demonstrated pioneering solutions to the problem of the user interface. The system displayed extremely large amounts of information to its operators using the then-new cathode ray tube; operators could then obtain additional information on aircraft tracks by selecting them with a light gun. Similar techniques are still in use today.
CG at General Motors (1961)

Figure 6. DAC-1 graphic console.

FIG. 1–8 Scale expansion, rotation, and partial views in a DAC-1 design exercise.
First video game (1962)

Watch Video: http://www.youtube.com/watch?v=Rmvb4Hktv7U
Computer games (the 80’s)

- The first generation of computer games were often text adventures or interactive fiction, in which the player communicated with the computer by entering commands through a keyboard.

- Later games combined textual commands with basic graphics, as seen in the SSI Gold Box games such as Pool of Radiance, or Bard's Tale for example.
Microchess was one of the first games for microcomputers which was sold to the public. First sold in 1977, Microchess eventually sold over 50,000 copies on cassette tape.
Atari consoles

- Frogger
- Space Invaders
- Donkey Kong
- Pac Man
Rise of PCs

- Ubiquity of PC’s and popularity of MS Windows coupled with success of 3D consoles and related titles fueled game industry further
- Nintendo’s Mario Brothers
- Atari’s Pitfall!
- Many others
PC Games

- Games like Wolfenstein 3D, Doom, Quake changed the gaming industry forever
- All you needed was a Personal Computer
Graphics hardware

- Graphics was mostly done on the CPU and there were early vendors such as 3dfx who developed specialized graphics hardware.
- Initially 3dfx developed cards for Atari, Nintendo 64, Playstation and Sega Saturn.
- Later, they developed and introduced their first graphics PCI expansion card - Voodoo Graphics PCI.
3D Graphics API

- Glide API was introduced by 3dfx to program the Voodoo Graphics PCI card
- Many other ways to program other imitation cards
- **OpenGL** emerged as the standard due to John Carmack’s decision to use it in Doom, Quake etc.
OpenGL

- OpenGL stands for Open Graphics Library
- Idea was to have a standard library that developers all over the world could use to program 3D graphics
- Vendors would then provide OpenGL capable drivers
Fixed Function Pipeline

- Key abstraction of real-time graphics
- Hardware used to look like this
- One chip/board per stage
- Fixed data flow through pipeline
Fixed Function Pipeline

- Everything is hardcoded
  - Lighting
  - Texture Mapping
- Hard to customize effects
  - Use parameters-based ‘customization’
- Widely used in most games from id Software
Companies such as Silicon Graphics (SGI) and Evans & Sutherland designed specialized and expensive graphics hardware.

The graphics systems developed by these companies introduced many of the concepts, such as vertex transformation and texture mapping, that we take for granted today.

These systems were very important to the historical development of computer graphics, but because they were so expensive, they did not achieve the mass-market success.
Commodity Graphics Hardware

- NVIDIA entered the graphics hardware industry and started competing with 3dfx and later ATI
- Developed 3D graphics accelerators
- Later known as **Graphics Processing Units (GPU)**
- Moved graphics pipeline pieces to hardware, thus providing huge speedups for graphics applications (mostly games!)
  - 200 Mhz clock with 1.5 Gigatexel/sec fill rate
What is a GPU?

- GPU stands for Graphics Processing Unit
- Specialized hardware for graphics
- Free the CPU from the burden of rendering
- Used in desktops, laptops, mobile devices and so on
The GPU

- GPU has evolved into a very powerful and flexible processor
  - Programmable using high level languages
  - Supports 32-bit and 64-bit floating point IEEE-754 precision
  - Capable of TeraFLOPS
FLOPS and Memory Bandwidth
Programmable Pipeline

- NVIDIA had moved all the possible units of the graphics pipeline to hardware
- Game developers needed more control over the output of their program
  - At the vertex and pixel level, they were constrained by the fixed function graphics pipeline
- Between 2001-2003, NVIDIA introduced programmable vertex shaders and pixel shaders
Programmable Pipeline

- Hardware used to look like this
  - Vertex, pixel processing became programmable
  - New stages added

*GPU architecture increasingly centers around shader execution*
New effects using Shaders

realistic materials  natural phenomena  non-photorealistic rendering

environment map  realistic surface  better anti-aliasing
New effects using Shaders

Complex Materials

Shadowing

Lighting Environments

Advanced Mapping
What is a shader?

- A program that runs on graphics hardware
- Used to be written in **assembly language**
- Allowed a user to perform simple math operations on vectors, matrices and textures such as add, multiply, dot product, sine, cosine etc.

- **Vertex shaders** are executed once per vertex
- **Geometry shaders** are executed once per primitive
- **Fragment/Pixel shaders** are executed once per pixel on the screen
OpenGL Pipeline

Line(p0, p1)
Triangle(p0, p1, p2)

Vertex Transformations
Geometry Assembly

Fragment Operations
Rasterization
Data Flow in the Graphics Pipeline

- Input Assembler
- Primitive Assembly
- Rasterization
- Framebuffer
- Vertex Processor
- Geometry Processor
- Fragment Processor
- Video Memory
Data Flow in the Graphics Pipeline

- Input Assembler
- Primitive Assembly
- Rasterization
- Fragment Processor
- Vertex Processor
- Geometry Processor
- Framebuffer

Video Memory

- Vertex Shader
- Geometry Shader
- Fragment Shader
**Vertex Shader**

- Executed once per vertex
- Transform input vertices

**Input attributes**
- Vertex normal
- Texture coordinates
- Colors
Geometry Shader

- Geometry composition

- Executed once per geometry

- Input primitives
  - Points, Lines, Triangles
  - Lines and Triangles with adjacency

- Output primitives
  - Points, line strips, triangle strips, quad strips
  - \([0, n]\) primitives outputted
Fragment/Pixel Shader

- Per-pixel (or fragment) composition
- Executed once per fragment
- Operations on interpolated values
  - Vertex attributes
  - User-defined varying variables
History of Shading Languages

- Renderman – Pixar, software used to create movies such as Toy Story
- Cg – NVIDIA developed the first commercial Shading Language
- HLSL – Microsoft & Nvidia (Xbox etc.)
- GLSL – SGI/3DLabs, Architecture Review Board (ARB)
- Stanford RTSL – Academic Shading Language
Fixed function tasks are bypassed

- **Vertex Tasks**
  - Vertex transformations
  - Normal transformation, Normalization
  - Lighting
  - Texture coordinate generation and transformation

- **Fragment Tasks**
  - Texture accesses
  - Fog
  - Discard fragment (culling)
Anatomy of GLSL

- **Built-in variables**
  - Always prefaced with `gl_`
  - Accessible to both vertex and fragment shaders

- **Uniform variables**
  - Matrices (i.e. `ModelViewMatrix`, `ProjectionMatrix`, inverses, transposes)
  - Materials (in `MaterialParameters` struct, ambient, diffuse, etc)
  - Lights (in `LightSourceParameters` struct, specular, position, etc)

- **Varying variables**
  - `FrontColor` for colors
  - `TexCoord [ ]` for texture coordinates
Anatomy of GLSL

- **Vertex Shaders**
  - Have access to vertex attributes such as `gl_Color`, `gl_Normal`, `gl_Vertex` etc.
  - Also write to special output variables such as `gl_Position`, `gl_PointSize` etc.

- **Fragment Shaders**
  - Have access to special input variables such as `gl_FragCoord`, `gl_FrontFacing`, etc.
  - Also write to special output variables such as `gl_FragColor`, `gl_FragDepth`, etc.
Structure of a shader

/**
 * Comments
 */

Global Definitions

void main (void)
{
    Body of the function
}

void main (void)
{

    // Pass vertex color to next stage
    gl_FrontColor = gl_Color;

    // Transform vertex position before passing it
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
Hello World – Geometry Shader

```c
# extension GL_EXT_geometry_shader4: enable
void main (void)
{
    // Iterate over all vertices in input primitive
    for(int i=0; i<gl_VerticesIn; ++i) {
        // Pass color and position to next stage
        gl_FrontColor = gl_FrontColorIn[i];
        gl_Position = gl_PositionIn[i];
        EmitVertex();
    }
    // Done with the input primitive
    EndPrimitive();
}
```
void main (void)
{
    // Pass fragment color
    gl_FragColor = gl_Color;
}
GLSL Basic Definitions

- Based on ANSI C with some C++ additions
- Vector types
  - Floats, integers and booleans
  - 2-, 3-, 4-components
- Vector components can be swizzled
- Matrix types
  - Only floats – 2x2, 3x3 and 4x4 matrices
- Texture types
  - 1-, 2-, 3-dimensional
  - Cube mapping
  - Shadow mapping

DATA TYPES
- float, vec2, vec3, vec4
- int, ivec2, ivec3, ivec4
- bool, bvec2, bvec3, bvec4
- mat2, mat3, mat4
- void
- sampler1D, sampler2D, sampler3D

VECTOR COMPONENTS
- component names may not be mixed across sets
- x, y, z, w
- r, g, b, a
- s, t, p, q
GLSL Basic Definitions

- **Type qualifiers**
  - Uniform
  - Attribute
  - Varying
- **Function qualifiers**
  - In, Out and in-out
  - Constant

**DATA TYPE QUALIFIERS**

**global variable declarations:**
- `uniform` input to Vertex and Fragment shader from OpenGL or application (READ-ONLY)
- `attribute` input per-vertex to Vertex shader from OpenGL or application (READ-ONLY)
- `varying` output from Vertex shader (READ/WRITE), interpolated, then input to Fragment shader (READ-ONLY)
- `const` compile-time constant (READ-ONLY)

**function parameters:**
- `in` value initialized on entry, not copied on return (default)
- `out` copied out on return, but not initialized
- `inout` value initialized on entry, and copied out on return
- `const` constant function input
GLSL Basic Definitions

- Built-in functions
  - sin, cos, tan
  - pow, exp, sqrt
  - **discard** keyword

- Built-in uniform variables
  - Modelview and Projection matrices
  - Texture and normal matrices

- Lighting and material parameters

- Fog and point parameters

```cpp
BUILT-IN UNIFORMS (7.5 p45) access=RO

uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];

uniform mat4 gl_ModelViewMatrixInverse;
uniform mat4 gl_ModelViewProjectionMatrixInverse;
uniform mat4 gl_ProjectionMatrixInverse;
uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];

uniform mat4 gl_ModelViewMatrixTranspose;
uniform mat4 gl_ModelViewProjectionMatrixTranspose;
uniform mat4 gl_ProjectionMatrixTranspose;
uniform mat4 gl_TextureMatrixTranspose[gl_MaxTextureCoords];
```
Input/Output – Vertex Shader

- **Standard attributes**:
  - `gl_Vertex`
  - `gl_Normal`
  - `gl_Color`
  - `gl_MultiTexCoord0`
  - ...

- **Built-in constants**:
  - `gl_MaxLights`, ...

- **Texture maps**

- **Vertex Shader**

- **User defined attributes**

- **User defined uniforms**
  - `gl_ModelViewMatrix`, ...

- **User-defined uniforms**

- **User defined varyings**

- **Special variables**
  - `gl_Position` *(shader must write)*

- **Standard varyings**
  - `gl_FrontColor`
  - `gl_TexCoord[]`
  - ...

- **Built-in uniforms**
Input/Output - Geometry Shader

- **Geometry Shader**
  - built-in constants: `gl_VerticesIn`, ...
  - texture maps
  - standard attributes:
    - `gl_PositionIn[gl_VerticesIn]`
    - `gl_FrontColorIn[gl_VerticesIn]`
    - `gl_TexCoordIn[gl_VerticesIn]`
    - ...
  - user defined attributes
  - built-in uniforms: `gl_ModelViewMatrix`, ...
  - user-defined uniforms
  - user defined varyings
  - special variables:
    - `gl_Position` *(shader must write)*
    - ...
    - `gl_FrontColor`
    - `gl_TexCoord[ ]`
    - ...
  - standard varyings
  - functions:
    - `void EmitVertex();`
    - `void EndPrimitive();`
Input/Output – Fragment Shader

- **standard varyings**:
  - `gl_Color`
  - `gl_FragCoord`
  - `gl_TexCoord[ ]`
  - ...

- **built-in constants**:
  - `gl_MaxTextureUnits`, ...

- **texture maps**

- **built-in uniforms**:
  - `gl_ModelViewMatrix`, ...

- **user-defined uniforms**

- **user-defined varyings**

- **special variables**:
  - `gl_FragColor` *(shader must write)*
  - ...

- **special keyword**
  - `discard;`
Simplifying Working with OpenGL

- Operating systems deal with library functions differently
  - compiler linkage and runtime libraries may expose different functions

- Additionally, OpenGL has many versions and profiles which expose different sets of functions
  - managing function access is cumbersome, and window-system dependent

- We use another open-source library, GLEW, to hide those details
An example – Animated Teapot

```c
#include <stdio.h>
#include <stdlib.h>

#include <GL/glew.h>
#include <GL/glut.h>

#include "textfile.h"

GLuint v,f,p;
float lpos[4] = {1,0.5,1,0};
float a = 0;
```

void reshape(int w, int h) {
    // Prevent a divide by zero, when window is too short
    // (you can't make a window of zero width).
    if (h == 0)
        h = 1;

    float ratio = 1.0 * w / h;

    // Reset the coordinate system before modifying
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Set the viewport to be the entire window
    glViewport(0, 0, w, h);

    // Set the correct perspective.
    gluPerspective(45, ratio, 1, 1000);
    glMatrixMode(GL_MODELVIEW);
}
void renderScene(void) {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();
    gluLookAt(0.0, 0.0, 5.0,
              0.0, 0.0, -1.0,
              0.0f, 1.0f, 0.0f);

    glLightfv(GL_LIGHT0, GL_POSITION, lpos);
    glRotatef(a, 0, 1, 1);
    glutSolidTeapot(1);
    a+=0.1;

    glutSwapBuffers();
}
void setShaders() {
    char *vs = NULL,*fs = NULL,*fs2 = NULL;
    v = glCreateShader(GL_VERTEX_SHADER);
    f = glCreateShader(GL_FRAGMENT_SHADER);

    vs = textFileRead("minimal.vert");
    fs = textFileRead("minimal.frag");
    const char * vv = vs;
    const char * ff = fs;
    glShaderSource(v, 1, &vv,NULL);
    glShaderSource(f, 1, &ff,NULL);
    free(vs);free(fs);

    glCompileShader(v);
    glCompileShader(f);

    printShaderInfoLog(v);
    printShaderInfoLog(f);

    p = glCreateProgram();
    glAttachShader(p,v);
    glAttachShader(p,f);

    glLinkProgram(p);
    printProgramInfoLog(p);

    glUseProgram(p);
}
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100,100);
    glutInitWindowSize(320,320);
    glutCreateWindow("MM 2004-05");
    glutDisplayFunc(renderScene);
    glutIdleFunc(renderScene);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(processNormalKeys);
    glEnable(GL_DEPTH_TEST);
    glClearColor(1.0,1.0,1.0,1.0);
    glEnable(GL_CULL_FACE);
    glewInit();
    if (glewIsSupported("GL_VERSION_2_0"))
        printf("Ready for OpenGL 2.0\n");
    else {
        printf("OpenGL 2.0 not supported\n");
        exit(1);
    }
    setShaders();
    glutMainLoop();
    return 0;
}
Vertex Shader

void main()
{
    // the following two lines provide the same result
    //   gl_Position = gl_ProjectionMatrix *
    //                  gl_ModelViewMatrix * gl_Vertex;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

Fragment Shader

void main()
{
    gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}

Show Demo
Example 2 – Rotating Cube

```c
#include <stdio.h>
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/glut.h>
#include "textfile.h"

GLint loc; float a = 0;
GLuint v,f,f2,p;
void reshape(int w, int h) {
    // Prevent a divide by zero, when window is too short (you cant make a window of zero width).
    if(h == 0)
        h = 1;
    float ratio = 1.0* w / h;
    // Reset the coordinate system before modifying
    glMatrixMode(GL_PROJECTION);  glLoadIdentity();
    // Set the viewport to be the entire window
    glViewport(0, 0, w, h);
    // Set the correct perspective.
    gluPerspective(45,ratio,1,100);
    glMatrixMode(GL_MODELVIEW);
}
```

Demo credits: http://www.lighthouse3d.com/tutorials/gls3-tutorial/color-shader/
void drawCube() {

    float hd = 1.0;

glColor3f(1,0,0);
gBegin(GL_QUADS);
    glVertex3f(-hd,-hd,-hd);
    glVertex3f(-hd,hd,-hd);
    glVertex3f(hd,hd,-hd);
    glVertex3f(-hd,hd,-hd);
    glEnd();

glColor3f(1,0,0);
gBegin(GL_QUADS);
    glVertex3f(-hd,-hd,-hd);
    glVertex3f(-hd,hd,-hd);
    glVertex3f(hd,hd,-hd);
    glVertex3f(-hd,hd,-hd);
    glEnd();

glColor3f(0,1,0);
gBegin(GL_QUADS);
    glVertex3f(-hd,-hd,hd);
    glVertex3f(-hd,hd,hd);
    glVertex3f(hd,hd,hd);
    glVertex3f(-hd,hd,hd);
    glEnd();

glColor3f(0,1,0);
gBegin(GL_QUADS);
    glVertex3f(-hd,-hd,hd);
    glVertex3f(-hd,hd,hd);
    glVertex3f(hd,hd,hd);
    glVertex3f(-hd,hd,hd);
    glEnd();

glColor3f(0,0,1);
gBegin(GL_QUADS);
    glVertex3f(-hd,-hd,hd);
    glVertex3f(-hd,hd,hd);
    glVertex3f(hd,hd,hd);
    glVertex3f(-hd,hd,hd);
    glEnd();

glColor3f(0,0,1);
gBegin(GL_QUADS);
    glVertex3f(-hd,-hd,hd);
    glVertex3f(-hd,hd,hd);
    glVertex3f(hd,hd,hd);
    glVertex3f(-hd,hd,hd);
    glEnd();

glColor3f(0,1,1);
gBegin(GL_QUADS);
    glVertex3f(hd,-hd,-hd);
    glVertex3f(hd,hd,-hd);
    glVertex3f(hd,hd,hd);
    glVertex3f(hd,-hd,hd);
    glEnd();
}
void renderScene(void) {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();
    gluLookAt(0.0,5.0,5.0,
              0.0,0.0,0.0,
              0.0f,1.0f,0.0f);

    glRotatef(a,0,1,0);

    drawCube();
    a+=1.0;

    glutSwapBuffers();
}

void setShaders() {

    char *vs = NULL,*fs = NULL;

    v = glCreateShader(GL_VERTEX_SHADER);
    f = glCreateShader(GL_FRAGMENT_SHADER);
    vs = textFileRead("color.vert");
    fs = textFileRead("color.frag");

    const char * vv = vs;  const char * ff = fs;

    glShaderSource(v, 1, &vv,NULL);
    glShaderSource(f, 1, &ff,NULL);

    free(vs);free(fs);
    glCompileShader(v);
    glCompileShader(f);

    p = glCreateProgram();
    glAttachShader(p,f);
    glAttachShader(p,v);

    glLinkProgram(p);
    glUseProgram(p);

    loc = glGetUniformLocation(p,"time");
}

int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(320, 320);
    glutCreateWindow("Lighthouse 3D");

    glutDisplayFunc(renderScene);
    glutIdleFunc(renderScene);
    glutReshapeFunc(changeSize);
    glutKeyboardFunc(processNormalKeys);

    glEnable(GL_DEPTH_TEST);
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glEnable(GL_CULL_FACE);

    glewInit();
    if (glewIsSupported("GL_VERSION_2_0")) {
        printf("Ready for OpenGL 2.0\n");
    } else {
        printf("OpenGL 2.0 not supported\n");
        exit(1);
    }

    setShaders();

    glutMainLoop();

    return 0;
}
void main()
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
void main()
{
    gl_FragColor = gl_Color;
}
Demo
Reading

- Chapter 18 – Fundamentals of CG
- Chapter 15 – OpenGL Programming Guide
Game developer salary survey

Survey highlights include:

- The average game programmer salary is $66,000.
- A technical director with 6 or more years experience earns an average of $104,000.
- Game artists earn an average of $61,000.
- A game designer with one year of experience earns an average of $52,000, with the highest salary reported at $300,000.
- Game producers earn an average of $76,000.
- Developer salaries are highest in California and Texas, where game development studios tend to cluster.
- Women in the game industry fare better than women in other industries, earning 89 cents on the dollar, exceeding the national average of 76 cents.