Instructor:
Mitja Hmeljak,
http://pages.iu.edu/~mitja
mitja@indiana.edu
Computer Graphics: Models and Architectures

Goals:

Learn the basic design of a graphics system

Introduce the polygon projection pipeline architecture

From the fixed pipeline to the programmable pipeline
So... how do we go about Image Formation?

Can we mimic the synthetic camera model to design graphics hardware software?

Application Programmer Interface (API)

Need only specify
- Objects
- Materials
- Viewer
- Lights

But how is the API implemented?
Some Possible Physical Approaches

**Ray tracing**: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
- Can handle global effects
  - Multiple reflections
  - Translucent objects
- Slow
- Must have whole data base available at all times

**Radiosity**: Energy based approach
- Very slow
A more Practical Approach

- Process objects one at a time, in the order they are generated by the application
- Consider only local lighting
- Implement in pipeline architecture

All steps can be implemented in hardware on the graphics card
Vertex Processing

Much of the work in the pipeline is in converting object representations from one coordinate system to another.

- Object coordinates
- Camera (eye) coordinates
- Screen coordinates

Every change of coordinates is equivalent to a matrix transformation.

Vertex processor also computes vertex colors.
Projection

*Projection* is the process that combines the 3D viewer with the 3D objects to produce the 2D image.

Perspective projections: all projectors meet at the center of projection.

Parallel projection: projectors are parallel, center of projection is replaced by a direction of projection.
Primitive Assembly

Vertices must be collected into geometric objects before clipping and rasterization can take place

- Line segments
- Polygons
- Curves and surfaces
Clipping

Just as a real camera cannot “see” the whole world, the virtual camera can only see part of the world or object space.

Objects that are not within this volume are said to be clipped out of the scene.
Rasterization

If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors.

Rasterizer produces a set of fragments for each object.

Fragments are “potential pixels”
- Have a location in frame buffer
- Color and depth attributes

Vertex attributes are interpolated over objects by the rasterizer.
Fragment Processing

Fragments are processed to determine the color of the corresponding pixel in the frame buffer.
Colors can be determined by texture mapping or interpolation of vertex colors.
Fragments may be blocked by other fragments closer to the camera.

Hidden-surface removal
The Programmer’s Interface

Programmer sees the graphics system through a software interface: the Application Programmer Interface (API)
API Contents

Functions that specify what we need to form an image
- Objects
- Viewer
- Light Source(s)
- Materials

Other information
- Input from devices such as mouse and keyboard
- Capabilities of system
Object Specification

Most APIs support a limited set of primitives including
Points (0D object)
Line segments (1D objects)
Polygons (2D objects)
Some curves and surfaces
  Quadrics
  Parametric polynomials

All are defined through locations in space or vertices
Example: Fixed-Pipeline OpenGL
Old style, e.g.
"desktop" OpenGL 1.x without extensions,
or OpenGL ES 1.x,
way before WebGL existed

```c
glBegin(GL_POLYGON)  
glVertex3f(0.0, 0.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glVertex3f(0.0, 0.0, 1.0);
glEnd();
```

*type of object*

*location of vertex*

*end of object definition*
Example: Programmable-Pipeline OpenGL (GPU based)

Put geometric data in an array:

```javascript
var points = [
    vec3(0.0, 0.0, 0.0),
    vec3(0.0, 1.0, 0.0),
    vec3(0.0, 0.0, 1.0),
    vec3(0.0, 0.0, 1.0),
];
```

Send array to GPU
Tell GPU to render as triangle
Camera Specification

Six degrees of freedom
  Position of center of lens
  Orientation
Lens
Film size
Orientation of film plane
Lights and Materials

Types of lights
  - Point sources vs distributed sources
  - Spot lights
  - Near and far sources

Color properties

Material properties
  - Absorption: color properties
  - Scattering
    - Diffuse
    - Specular
SGI and GL

Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the fixed pipeline in hardware (1982)

To access the system, application programmers used a library called GL

With GL, it was relatively simple to program three dimensional interactive applications
OpenGL

The success of **GL** lead to **OpenGL** (1992), a platform-independent API that was:

- Easy to use
- Close enough to the hardware to get excellent performance
- Provided easy access to rendering
- Omitted windowing and input, to avoid operating system and window system dependencies
OpenGL Evolution

Originally defined by the *OpenGL Architectural Review Board (ARB)* (members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM, ...) now it's defined by the "Kronos Group"

OpenGL has been relatively stable (through version 2.5) :
- Backward compatible
- Evolution reflected new hardware capabilities
  - 3D texture mapping and texture objects
  - Vertex and fragment programs

Allows platform specific features through extensions
Modern OpenGL

- Performance is achieved by using GPU rather than CPU
- Control GPU through programs called shaders
- The application’s job on the CPU is to send data to GPU
- The GPU does all rendering
before OpenGL ES 2.0, the fixed pipeline was programmed using **Immediate Mode Graphics**

Geometry specified by vertices

- Locations in space (2 or 3 dimensional)
- Points, lines, circles, polygons, curves, surfaces

"**Immediate mode graphics**" works like this:

- Each time a vertex is specified in application, its location is sent to the GPU, using `glVertex`
- It creates bottleneck between CPU and GPU
- It has been removed from APIs, starting from versions OpenGL 3.1 and OpenGL ES 2.0:
- Immediate Mode Graphics does NOT EXIST in OpenGL ES >= 2.0 and WebGL!
more modern approach, started even before programmable pipeline: *Retained Mode Graphics*

- Put all vertex attribute data (coordinates, etc.) in an array

- Send array to GPU to be rendered immediately

- Almost OK, but there is a problem: we might have to send the array from CPU to GPU each time we need to render those vertices again...

- ...better to send array over and store it on the GPU for multiple renderings
modern APIs are totally shader-based

There are no "default" shaders

Each application must provide both a vertex and a fragment shader

There is no immediate mode

Few state variables

Most OpenGL 2.5 functions have been deprecated

Backward compatibility is not required

Exists a compatibility extension
OpenGL Versions

**OpenGL ES**
- Embedded systems
  - Version 1.0 equivalent to simplified OpenGL 2.1 (fixed pipeline)
  - Version 2.0 equivalent to simplified OpenGL 3.1 (shader based)

**WebGL**
- Javascript implementation of ES 2.0
- Supported on newer browsers

**OpenGL 4.1, 4.2, …..**
- Adds new shaders for geometry, tessellation, general computation
Software Organization the way it was, before the programmable pipeline

OpenGL (even when the only version was the "desktop" version) always had to rely on other libraries:
A Simple OpenGL Program, the way it was

For example, to draw a square on a solid background
A Simple OpenGL Program, *it used to be so easy* ...

```c
#include <GL/glut.h>

void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_QUAD);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
    glEnd();
}

int main(int argc, char** argv)
{
    glutCreateWindow("simple");
    glutDisplayFunc(mydisplay);
    glutMainLoop();
}
```
Why won't that code run anymore?
What happened to OpenGL?

Most "old style" OpenGL functions have been deprecated
- immediate \(\rightarrow\) retained mode
- compute on CPU \(\rightarrow\) make use of GPU

Makes heavy use of state variable default values that no longer exist
- Viewing
- Colors
- Window parameters

However, the processing loop is similar
In today's graphics pipeline, high performance is achieved by using GPU rather than CPU whenever possible.

The CPU-based application controls the GPU through programs called shaders.

Application’s (CPU-based) job is to send data to GPU.

The GPU does all rendering.
In sample programs there is a `render()` function which is an event listener or callback function.

The CPU program will have a `render callback`.

For a static application we need only execute the render function once.

In a dynamic application, the render function can call itself recursively but each redrawing of the display must be triggered by an event.
Lack of Object Orientation

Alas, all these versions of OpenGL are still not object oriented, so that there are multiple functions for a given logical function...

Example: sending values to shaders

- `glUniform3f`
- `glUniform2i`
- `glUniform3fv`

The "underlying" storage mode is the same
OpenGL ES function format:

\textbf{gl\textunderscore Uniform3f(location, x, y, z)}

- \textit{function name}
- \textit{dimension}
- \textit{location, x, y, z} are variables

\textbf{gl\textunderscore Uniform3fv(location, count, p)}

- \textit{p} is an array with \textit{count} elements