CSC 8470
Computer Graphics

What is Computer Graphics?

For us, it is primarily the study of how pictures can be generated using a computer:

But it also includes:

- software tools used to make pictures
- hardware for efficient graphics processing
- input devices for interacting with these pictures
- output devices for displaying the pictures

What You Will Learn

- How to produce simple images
- How graphics APIs and graphics hardware work
- Graphics APIs (Basic OpenGL)
- Modeling (Basic Blender)
- The math behind them all

- Balance between theory and practice

What You Will Not Learn

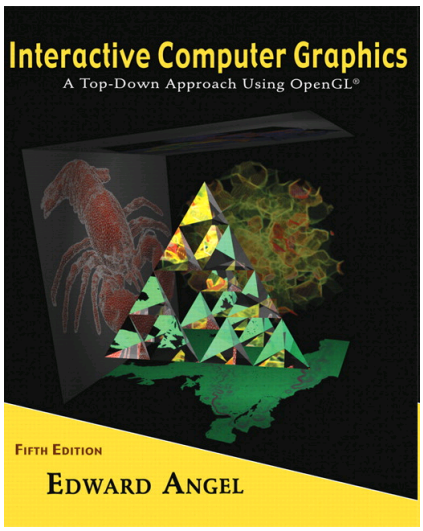
- Other software packages
  - CAD-CAM
  - Photoshop and other painting tools
- Artistic skills
- Game design

What You Need to Know

- Linear algebra and trigonometry
  - Brief review in class
- C-programming
  - Not hard to learn if you have not used it yet.

Resources

- Recommended:
- Online material
  - References on the class website



Grading

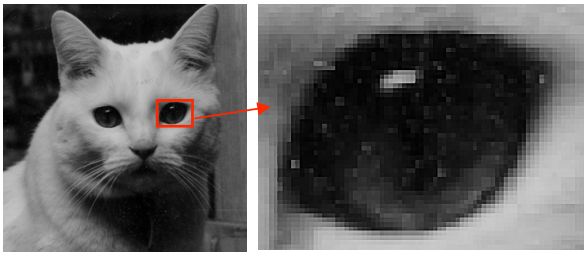
- Assignments (40%)
  - Pencil and paper
  - Clean, readable, working code
  - On time (no late submission)
  - **First assignment is available now**
- Exams (30%)
- Project (30%)
- No makeup assignments or exams
- Lecture attendance is ***mandatory***

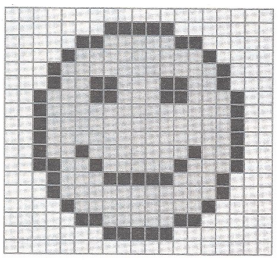
Create Your Own Project

- New idea or extension of previous project
- Suggestions throughout the semester
- We'll review your proposal to make sure the scope is appropriate

Graphics Systems Basics

What is a digital image?





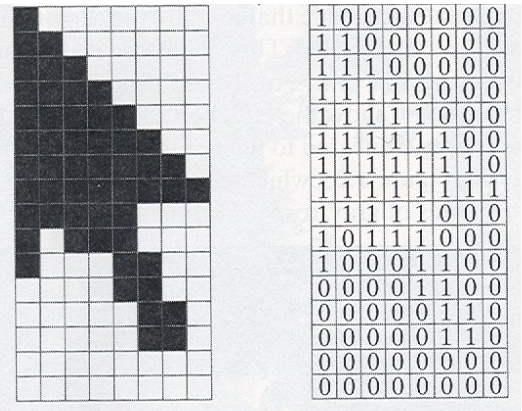
Pixel: picture element

Raster image: image stored as an array of numerical values, one per pixel.

2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	7
2	2	2	2	2	7	7	1
2	2	2	2	7	1	1	1
2	2	2	7	1	1	1	1
2	2	2	7	1	1	7	7

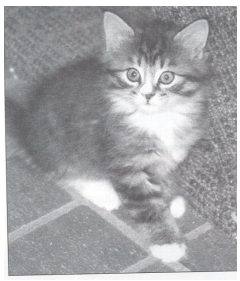
Pixel Depth

Pixel depth = number of bits allocated per pixel color.

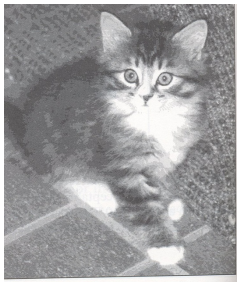


1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	1	0	0	0
1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1
1	1	1	1	1	0	0	0
1	0	1	1	1	0	0	0
1	0	0	0	1	1	0	0
0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	0
0	0	0	0	0	1	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

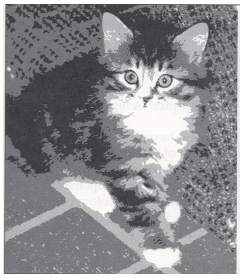
Effect of Pixel Depth



____ bits



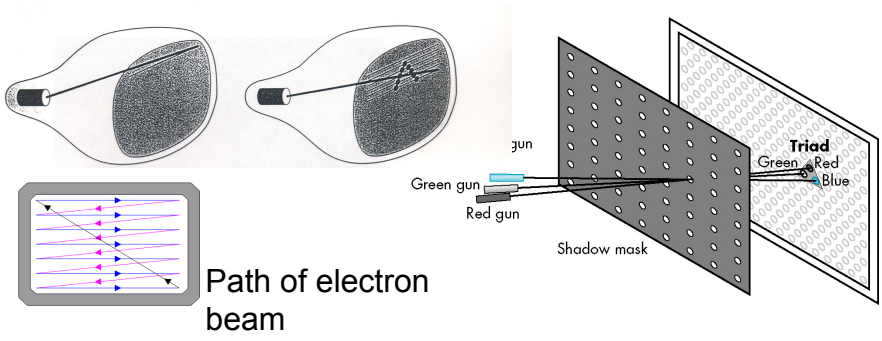
____ bits



____ bits

Banding: “lakes” of uniform color.

Raster Displays



Path of electron beam

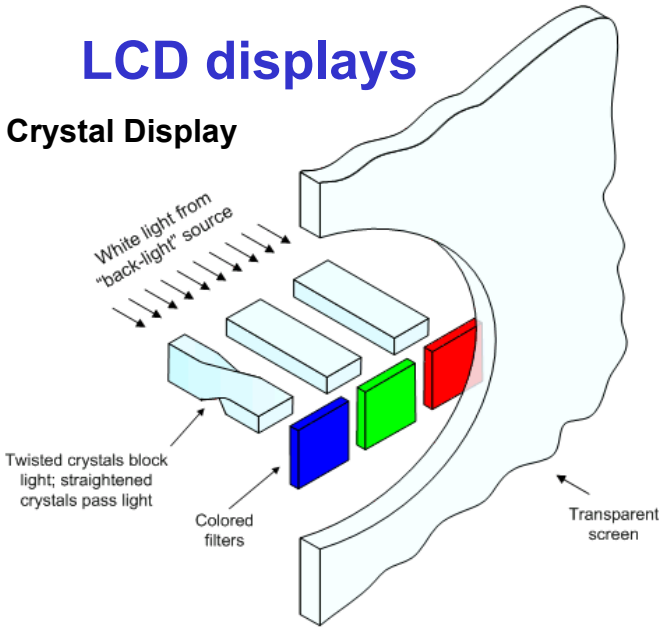
Display surface: Dotted with material that emits light when stimulated by electron beam. Beam must repeatedly stimulate dots row by row, because intensity fades when not stimulated.

Refresh rate: number of times the beam passes over the dots (~60-80/s)

Resolution: number of pixels

LCD displays

LCD: Liquid Crystal Display



High-level representation of one pixel in an LCD

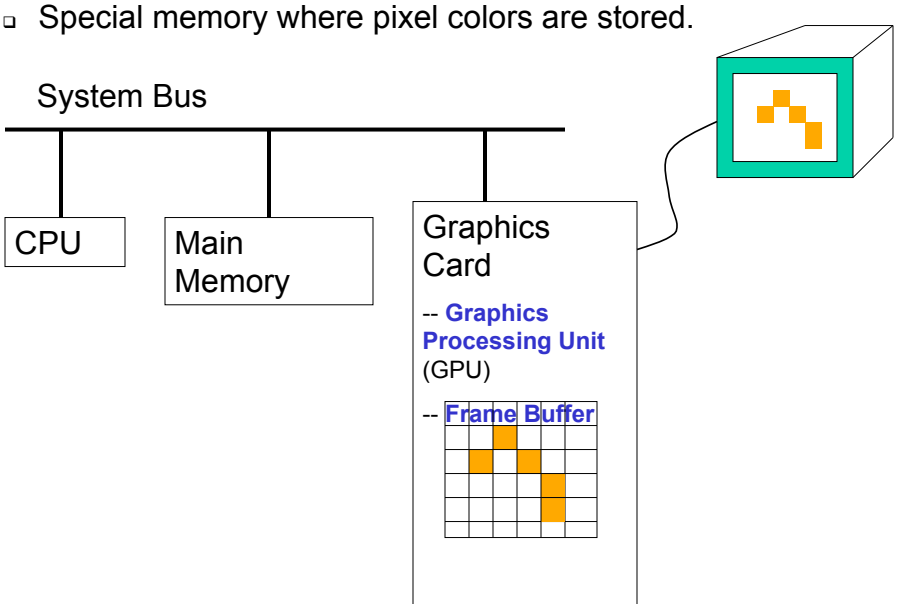
Color Raster Images

Pixel color is encoded as ordered triple.

Color value (R,G,B)	Displayed
(0, 0, 0)	Black
(0, 0, 1)	Blue
(0, 1, 0)	Green
(1, 0, 0)	Red
(1, 0, 1)	Magenta
(1, 1, 0)	Yellow
(0, 1, 1)	Cyan
(1, 1, 1)	White

Frame Buffer

- Special memory where pixel colors are stored.

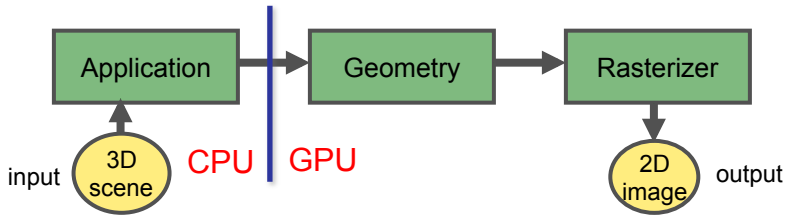


The Graphics Rendering Pipeline

The Graphics Rendering Pipeline

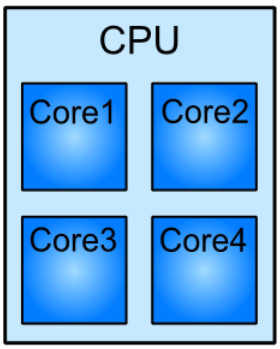
What is it?

The engine that creates 2D images from 3D scenes.

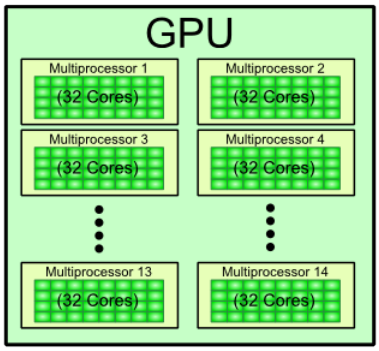


- CPU: Central Processing Unit
- GPU: Graphics Processing Unit

CPU/GPU Comparison



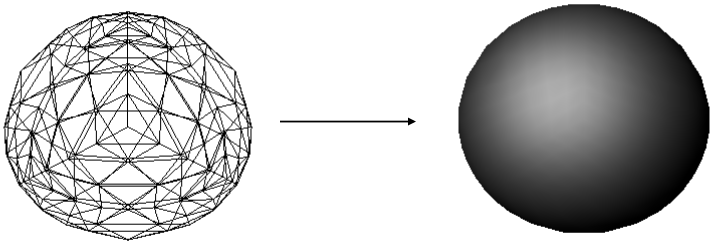
Few cores (4-8) optimized for serial processing.



Hundreds of specialized cores optimized for parallel geometric calculations and rendering.

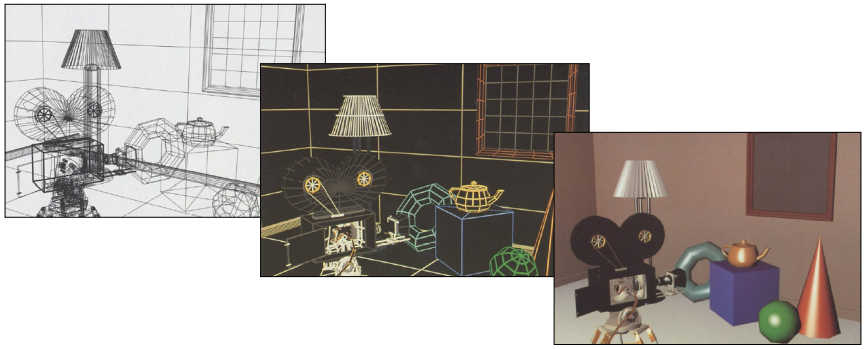
Rendering Primitives

- GPUs can render points, lines, triangles.
- A surface is thus an approximation by a number of such primitives.

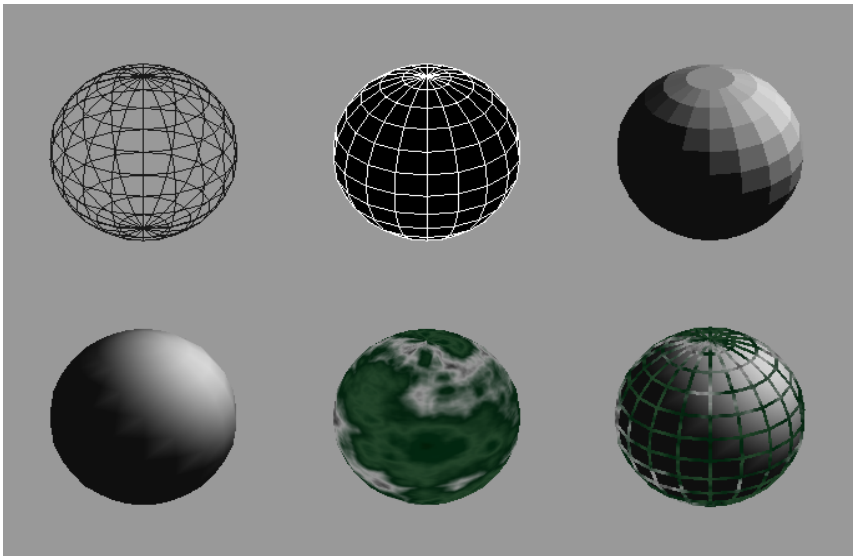


What is a 3D scene?

- A 3D scene is:
  - Geometry (triangles, lines, quads)
  - Material properties of geometry
  - Light sources (one or more)
  - Textures (images to glue onto the geometry)

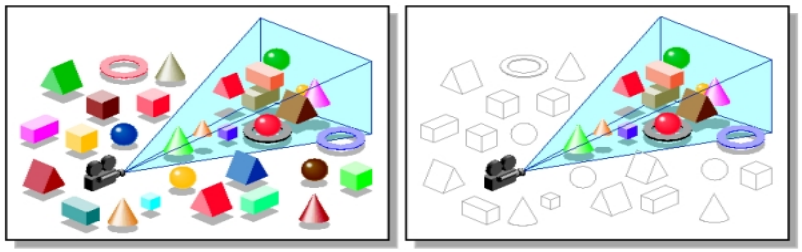


Wireframe to Texture Mapped



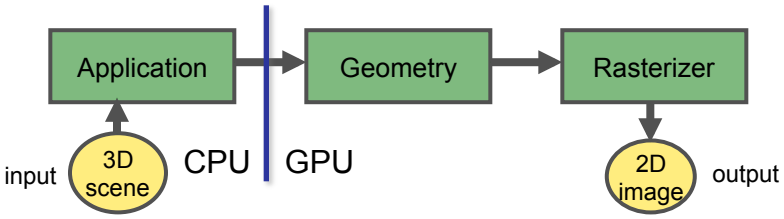
Virtual Camera

- To take a picture you need a camera
  - Decide what should end up in the final image

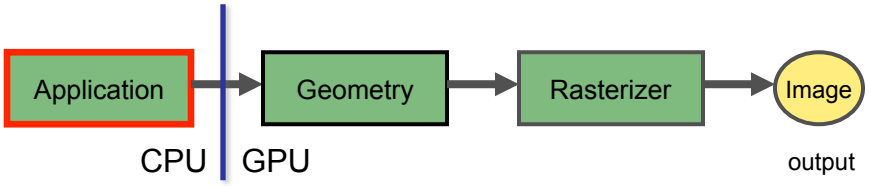


- Create image of geometry inside camera view

Back to Rendering Pipeline …

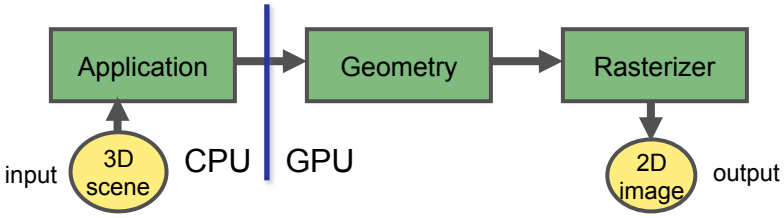


Rendering Pipeline – Application

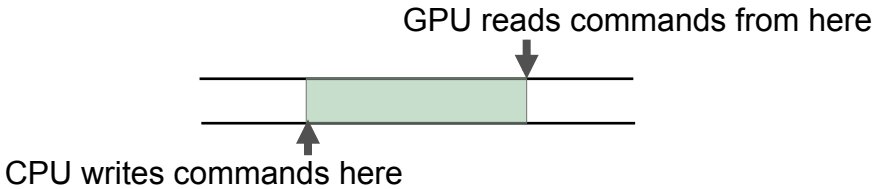


- Executed on the CPU
  - The programmer decides what happens here
- Examples:
  - Collision detection
  - Speed-up techniques
  - Animation
- Most important task:
  - send rendering primitives (e.g. triangles) to the GPU

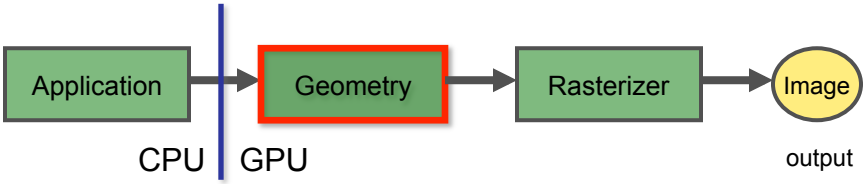
CPU – GPU Communication



- The CPU and GPU work in parallel with each other
- CPU thread communicates with GPU thread through a command buffer:

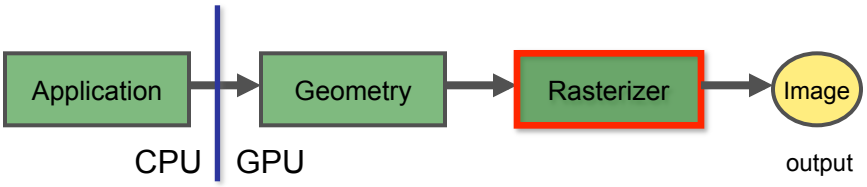


Rendering Pipeline – Geometry

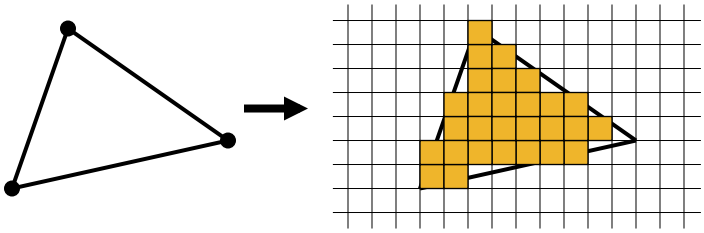


- The geometry stage does per-vertex operations:
  - Move objects (matrix multiplication)
  - Move the camera (matrix multiplication)
  - Compute lighting at vertices of triangle
  - Project onto screen (3D to 2D)
  - Animate objects

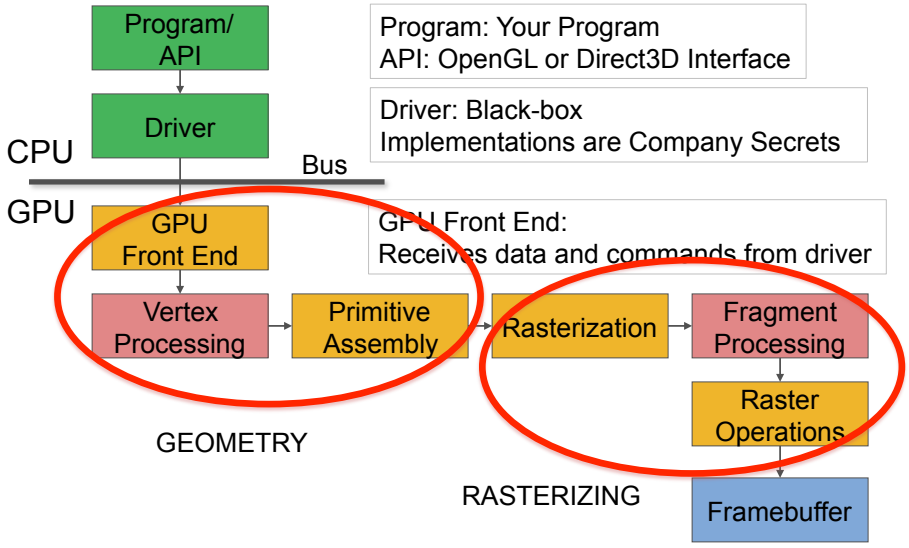
Rendering Pipeline – Rasterizer



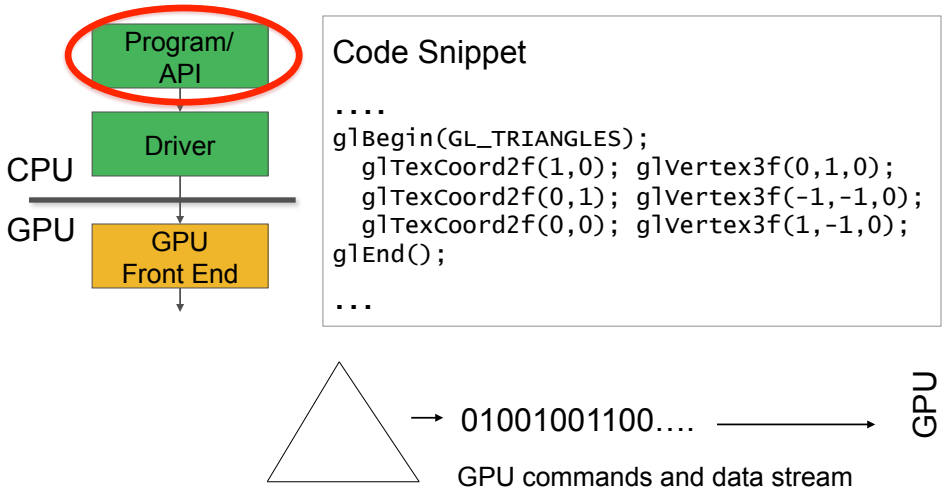
- The rasterizer stage does per-pixel operations:
  - Turn geometry into visible pixels on screen
  - Add textures
  - Resolve visibility (Z-buffer)



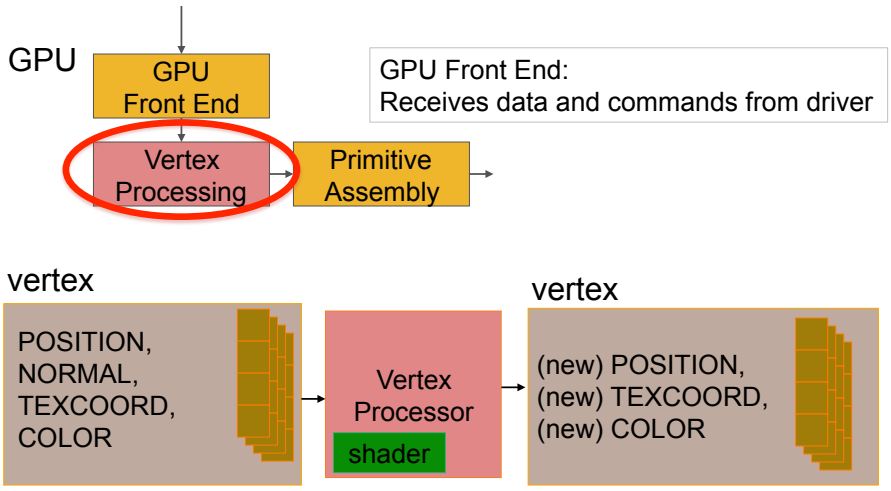
Rewind! Let's take a closer look …



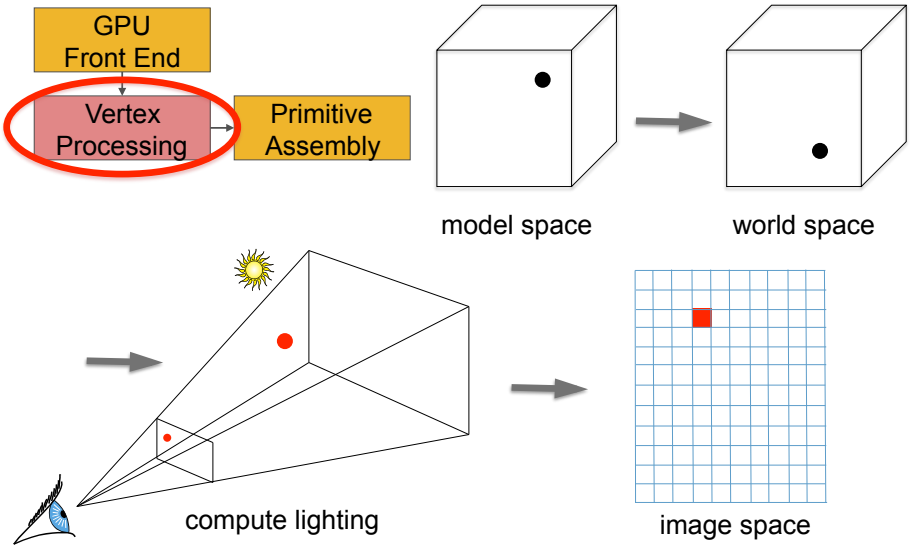
Graphics Pipeline – API Example



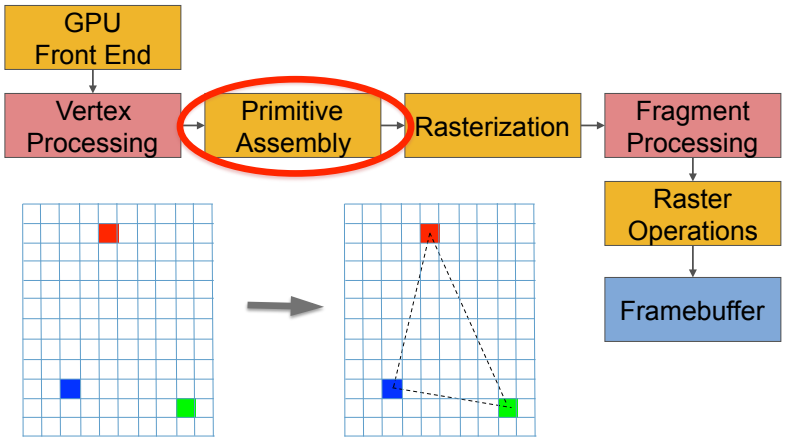
GPU Pipeline – Vertex Shading



GPU Vertex Processing Example

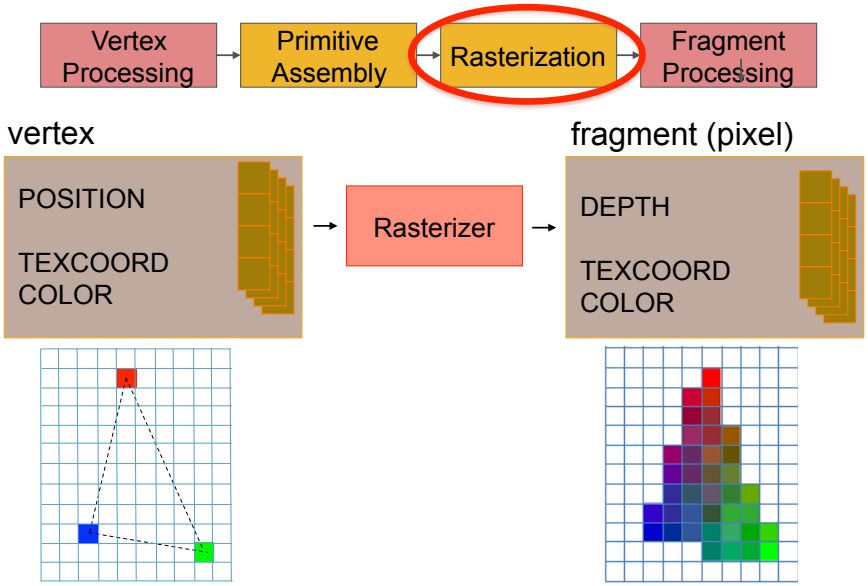


GPU Pipeline – Primitive Assembly

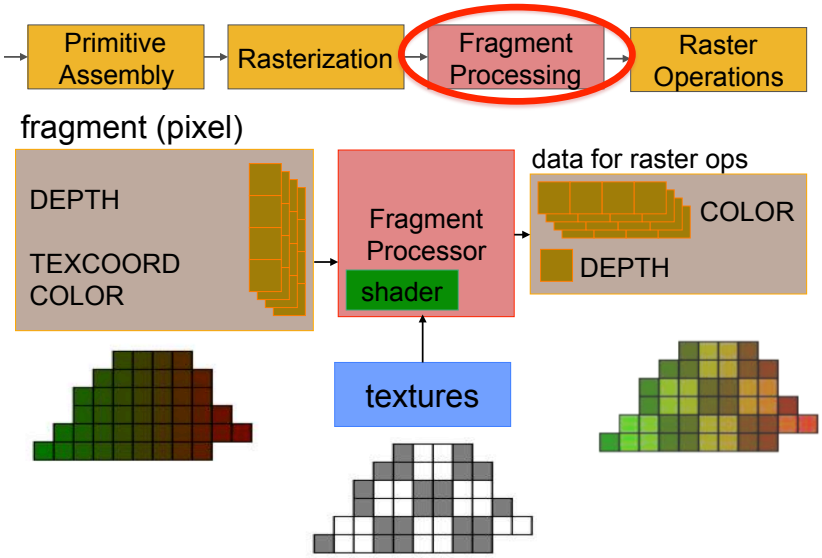


Compiles Vertices into Points, Lines and/or Polygons
Links elements so that the rasterizer can process them

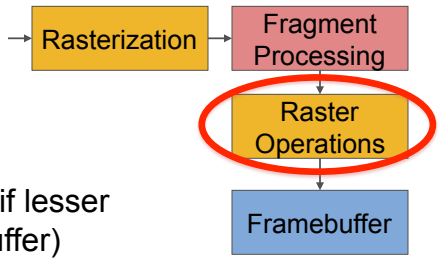
GPU Pipeline – Rasterization



GPU Pipeline – Fragment Process



GPU Pipeline – Raster Operations



- Depth Checking
  - Check framebuffer to see if lesser depth already exists (Z-Buffer)
  - Limited Programmability
- Blending
  - Use alpha channel to combine colors already in the framebuffer
  - Limited Programmability

Hands-On Session

- A cross-platform Application Programming Interface (API) for producing 2D and 3D computer graphics
- Developed by Silicon Graphics Inc. (1992)
- See: http://www.opengl.org

OpenGL Graphics Programming

- OpenGL
  - Industry standard API for 3D graphics
  - Cross-platform – Windows, MAC, Unix
  - Supported on all hardware platforms
- OpenGL API (gl)
  - core of OpenGL API
- OpenGL Utility API (glu)
  - additional functionality built on top of GL

GLUT

- GLUT = OpenGL Utility Toolkit
- Freeware library for simplifying interacting with windowing systems

GLUT Basics

- Application Structure
  - Configure and open window
  - Initialize OpenGL state
  - Register input callback functions
    - render
    - resize
    - input: keyboard, mouse, etc.
  - Enter event processing loop

Sample Program

keywords

```
int main( int argc, char** argv )
{
  int mode = GLUT_RGB | GLUT_DOUBLE;
  glutInitDisplayMode( mode );
  glutCreateWindow( argv[0] );
  init();
  glutDisplayFunc( display );
  glutReshapeFunc( resize );
  glutKeyboardFunc( key );
  glutIdleFunc( idle );
  glutMainLoop();
}
```

OpenGL Initialization

- Set up whatever state you are going to use

```
void init( void )
{
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glClearDepth( 1.0 );

  glEnable( GL_LIGHT0 );
  glEnable( GL_LIGHTING );
  glEnable( GL_DEPTH_TEST );
}
```

GLUT Callback Functions

- Routine to call when something happens
  - window resize or redraw
  - user input
  - animation
- "Register" callbacks with GLUT

```
glutDisplayFunc( display );
glutIdleFunc( idle );
glutKeyboardFunc( keyboard );
```

GLUT Rendering Callback

- Do all of your drawing here

```
glutDisplayFunc( display );
```

```
void display( void )
{
  glClear( GL_COLOR_BUFFER_BIT );
  glBegin( GL_TRIANGLE_STRIP );
    glVertex3fv( v[0] );
    glVertex3fv( v[1] );
    glVertex3fv( v[2] );
    glVertex3fv( v[3] );
  glEnd();
  glutSwapBuffers();
}
```

GLUT User Input Callback

- Process user input

```
glutKeyboardFunc( keyboard );
```

```
void keyboard( char key, int x, int y )
{
  switch( key ) {
    case 'q' :
    case 'Q' :
                exit( EXIT_SUCCESS );
                break;

    case 'r' :
    case 'R' :
                rotate = GL_TRUE;
                break;
  }
}
```

Hands-on Activities

- Soon you will write your own OpenGL program, but this time you'll copy an existing program (squares.cpp) from the course website.

- Get the code running and show it to your instructor.

OpenGL Elementary Rendering

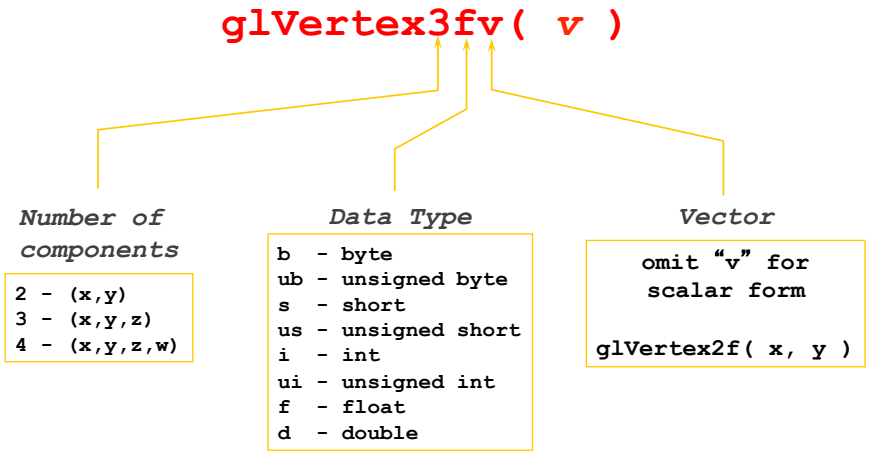
- OpenGL Geometric Primitives
- Managing OpenGL State
- Sierpinski's Triangle

Points, Lines, Polygons

Vertices

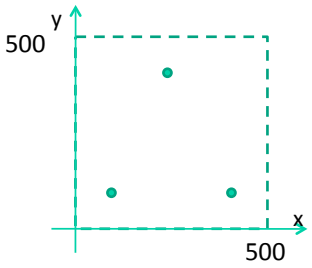
- A vertex is a location in the plane
  - Specified by its x and y coordinates
  - Used to define geometric primitives
  - The simplest geometric primitive is a point
- General form: glVertex*
  - Examples:
    - glVertex2i(int x, int y);
    - glVertex3f(float x, float y, float z);
    - glVertex3fv(float vcoord);

glVertex*

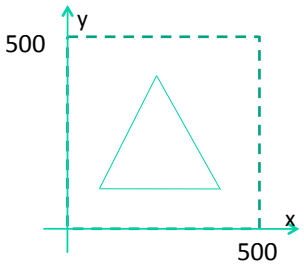


Simple Examples

```
glBegin( GL_POINTS );
    glVertex2i(100, 100);
    glVertex2i(400, 100);
    glVertex2i(250, 400);
glEnd();
```




```
glBegin( GL_LINE_LOOP );
    glVertex2i(100, 100);
    glVertex2i(400, 100);
    glVertex2i(250, 400);
glEnd();
```

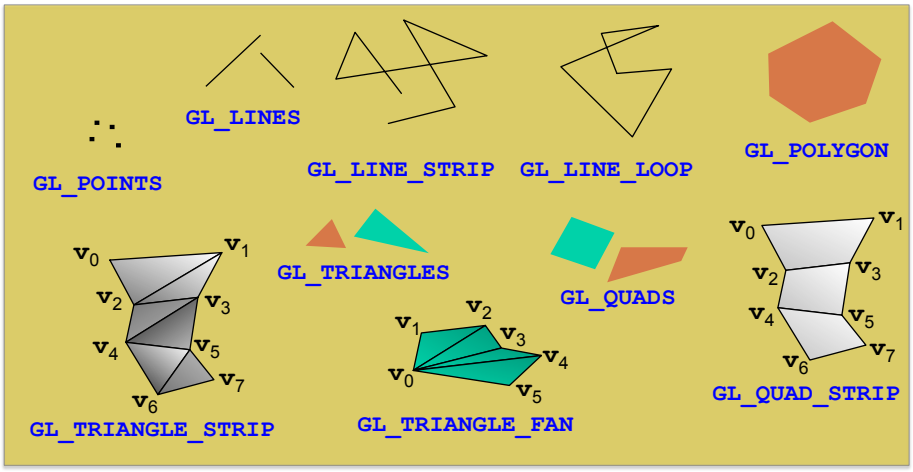


Many Questions

- How to interpret the values of x, y?
- In what units are they? Feet, meters or pixels?
- Where is the origin?
- Where on the screen does our image appear?
- How large will the image be?
- In what color are we drawing?

OpenGL Geometric Primitives

- All geometric primitives specified by vertices



Specifying Geometric Shapes

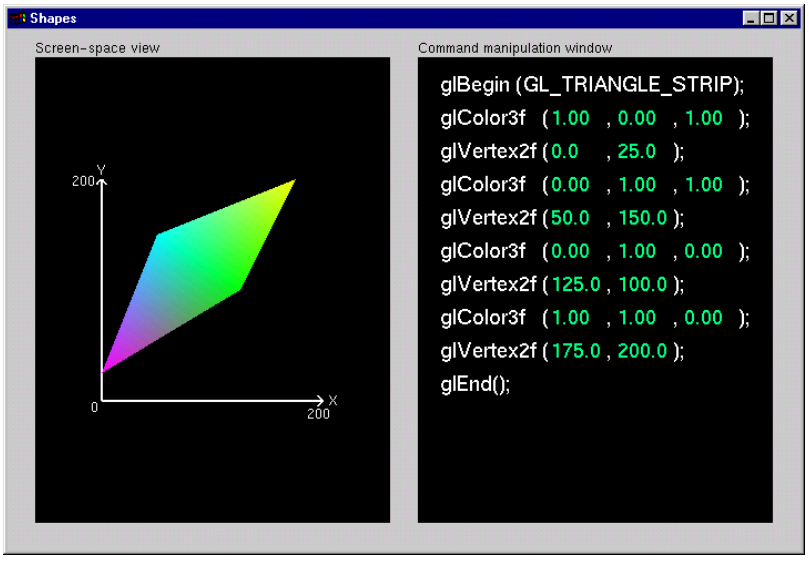
- Primitives are specified using

```
glBegin( primType );
glEnd();
```

  - *primType* determines how vertices are combined

```
GLfloat red, greed, blue;
Glfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i ) {
  glColor3f( red, green, blue );
  glVertex3fv( coords );
}
glEnd();
```

Shapes Tutorial



OpenGL's State Machine

- All rendering attributes are encapsulated in the OpenGL State
  - rendering styles
  - shading
  - Lighting, etc.

- Once an attribute is set, it stays that value until you later change it.

OpenGL Attribute Examples

```
glColor3f( 1.0, 0.0, 0.0 );

glClearColor( 1.0, 1.0, 1.0, 1.0 );

glPointSize( 4.0 );

glLineWidth( 2.0 );
```

Hands-On Session

- The primitives.cpp Program
- Compile, Run & Play, eg:
  - Change point size to 8
  - Change `GL_POINTS' to:
    - GL_LINES
    - GL_LINE_STRIP
    - GL_LINE_LOOP
    - GL_POLYGON
    - GL_QUADS
    - GL_TRIANGLES
    - GL_QUAD_STRIP
    - GL_TRIANGLE_STRIP
    - GL_TRIANGLE_FAN

The Sierpinski Gasket

Algorithm:

Start with 3 corner vertices v1, v2, v3 and one interior point p

Repeat the following

Pick an arbitrary vertex vi

Update p to be the midpoint of (p,vi)

N times, for some N

Hands-on Session

- The gasket.cpp Program
- Compile, Run & Play
- Change 'GL_POINTS' to GL_LINES
- Comment out certain lines, e.g.
  - glClear(*GL_COLOR_BUFFER_BIT*);
    (and resize your window)
- What happens when you resize your window?

Sierpinski Gasket -Triangle Bisection

Algorithm:

Start with a triangle (3 sides)
Repeat the following
    Compute side midpoints a, b, c
    Draw triangle abc
    Recurse on all but inner triangle abc

Hands-on Session

- Save a copy of your gasket.cpp code
- Alter the code for the gasket to generate the Sierpinski gasket using triangle bisection