CSCI 420 Computer Graphics
Lecture 3

Graphics Pipeline

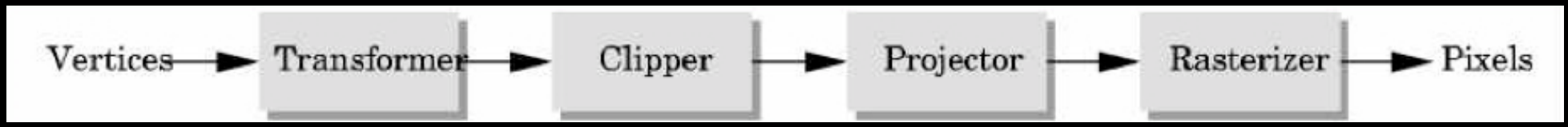
Graphics Pipeline
Primitives: Points, Lines, Triangles
[Angel Ch. 2]

Oded Stein
University of Southern California

How a triangle makes it from your mind to the screen

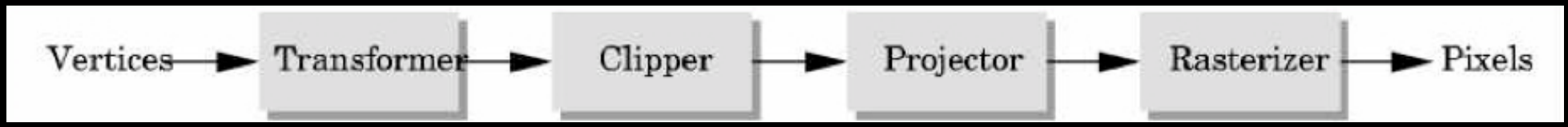
- The last class was a little bit abstract.
- Bits of code without context
- This is how OpenGL actually works (in the abstract)
- Will look at the whole pipeline today, and then bits of it in the next few classes.

Graphics Pipeline



Primitives+ material properties	Translate Rotate Scale	Is it visible on screen?	3D to 2D	Convert to pixels	Shown on the screen (framebuffer)

Graphics Pipeline



Primitives+ material properties	Translate Rotate Scale	Is it visible on screen?	3D to 2D	Convert to pixels	Shown on the screen (framebuffer)

- Not shown here:
  - colors
  - attributes
  - effects
  - shader programs

The Framebuffer

- Special memory on the graphics card
- Stores the current pixels to be displayed on the monitor
- Monitor has no storage capabilities
- The framebuffer is copied to the monitor at each refresh cycle

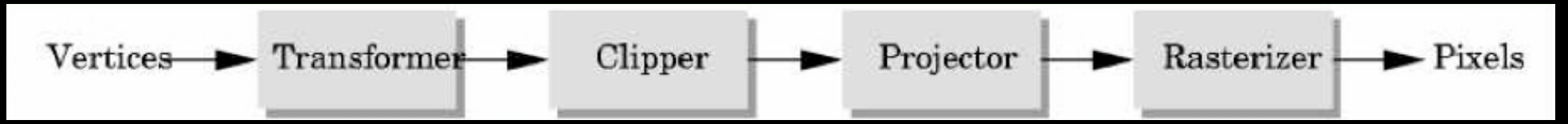
Rendering with OpenGL

- Application generates the geometric primitives (polygons, lines)
- System draws each one into the framebuffer
- Entire scene redrawn anew every frame
- Compare to: off-line rendering (e.g., Pixar Renderman, ray tracers)

Rendering with OpenGL

- Application generates the geometric primitives (polygons, lines)
  - Video game
  - Interactive application (CAD, etc.)
  - High-performance computing
- Written in any language on the CPU.

The pipeline is implemented by OpenGL, graphics driver and the graphics hardware

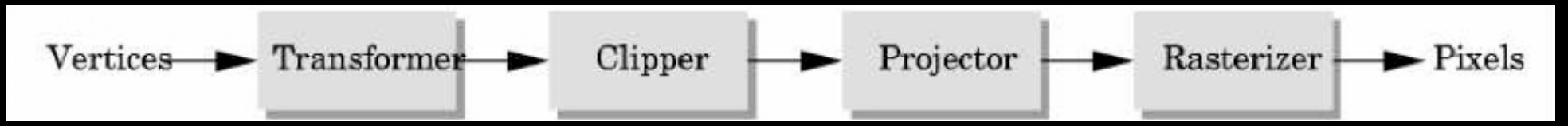


OpenGL programmer does not need to implement the pipeline.

However, pipeline is reconfigurable

➔ “shaders”

Graphics Pipeline



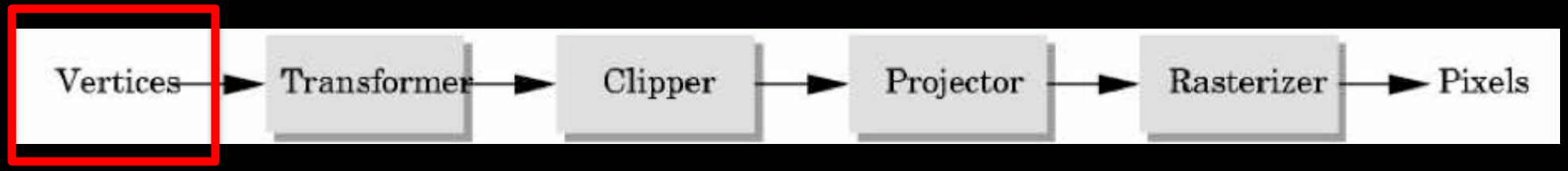
- Efficiently implementable in hardware (but not in software)

- Each stage can employ multiple specialized processors, working in parallel, buses between stages

- #processors per stage, bus bandwidths are fully tuned for typical graphics use

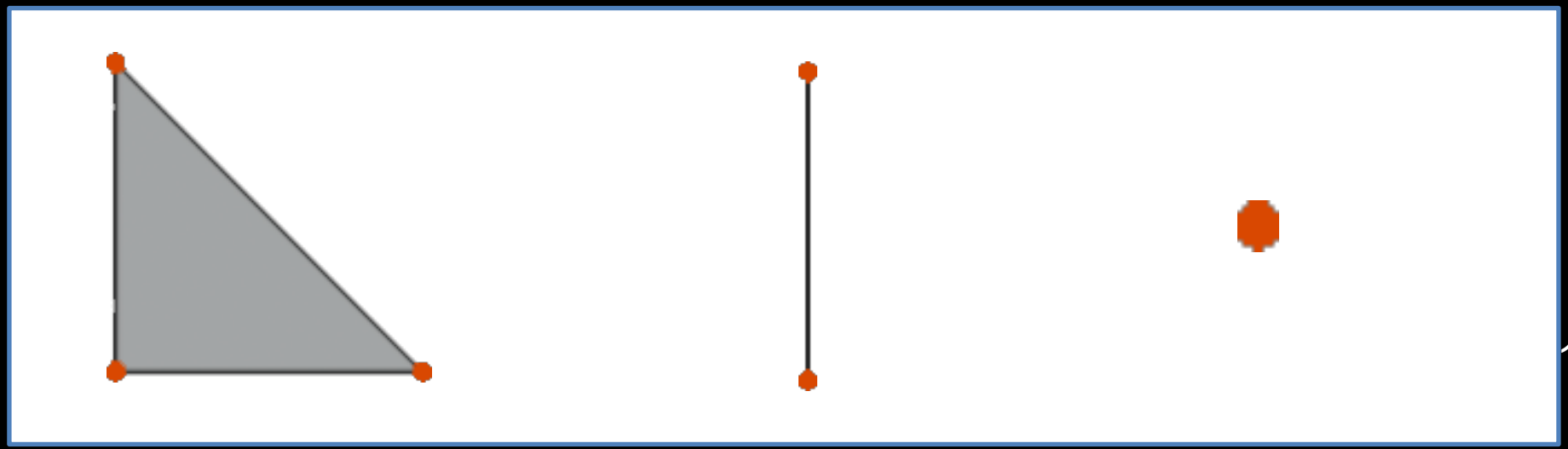
- Latency vs throughput

Vertices

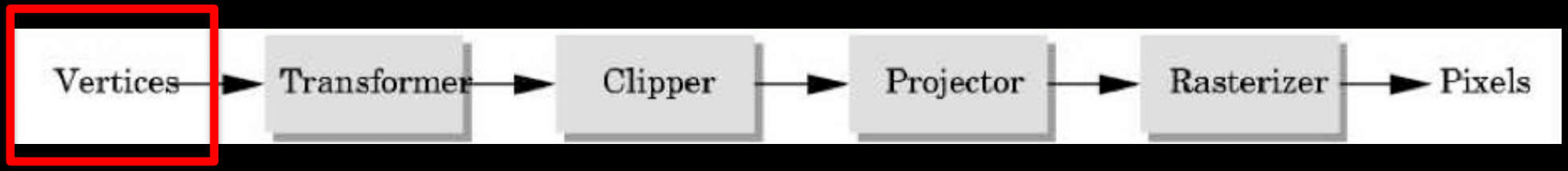


- What is a vertex?

- A vertex is a point (sort of)
  - A point that is a part of a larger geometric object



Vertices

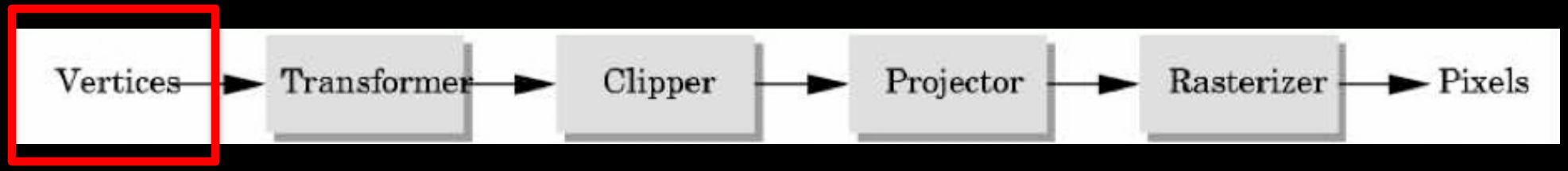


- What is a vertex?

- A vertex is a point (sort of)
  - A point that is a part of a larger geometric object

- The vertex is the only geometric object that lives on the GPU.
  - Everything else is a per-vertex attribute or a global attribute.

Vertices (compatibility profile)

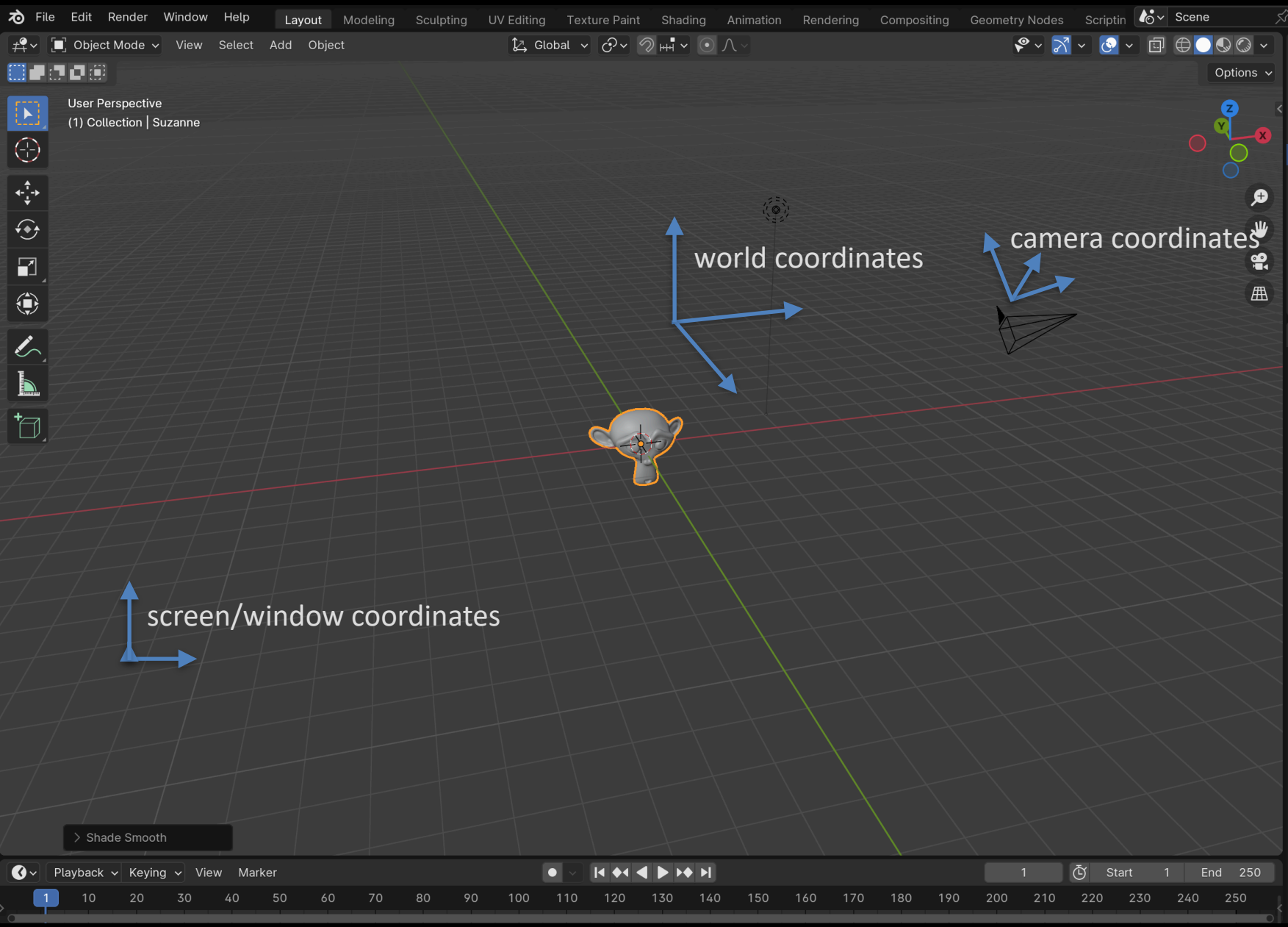


- Vertices in world coordinates

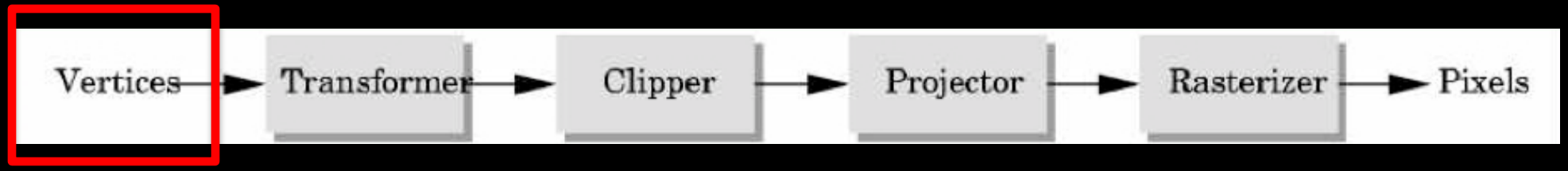
```
void glVertex3f(GLfloat x, GLfloat y, GLfloat z);
```

  – Vertex (x, y, z) is sent down the pipeline.
  – Function call then returns.
- Use GL_type for portability and consistency
- glVertex{234}{sfid}[v](*TYPE coords*)

Coordinate systems

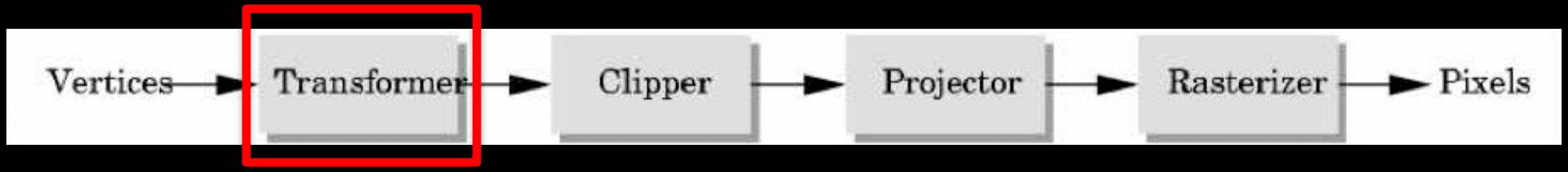


Vertices (core profile)



- Vertices in world coordinates
- Store vertices into a Vertex Buffer Object (VBO)
- Upload the VBO to the GPU during program during program initialization (before rendering)
- OpenGL renders directly from the VBO

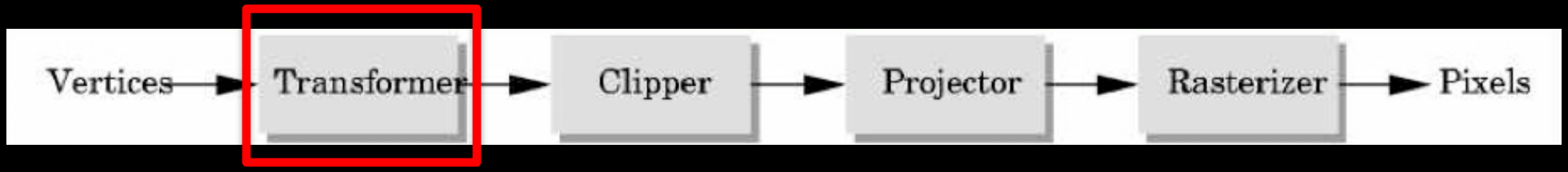
Transformer



- The OpenGL camera is in one place only.
  - Looks at the xy plane

- If you would like to look at the world from any other direction, you must rotate your world to fit.

Transformer (compatibility profile)

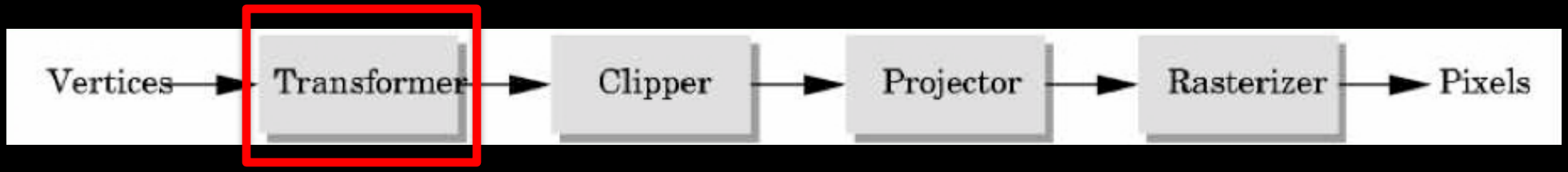


- Transformer in world coordinates
- Must be set before object is drawn!

```
glRotatef(45.0, 0.0, 0.0, -1.0);
glVertex2f(1.0, 0.0);
```

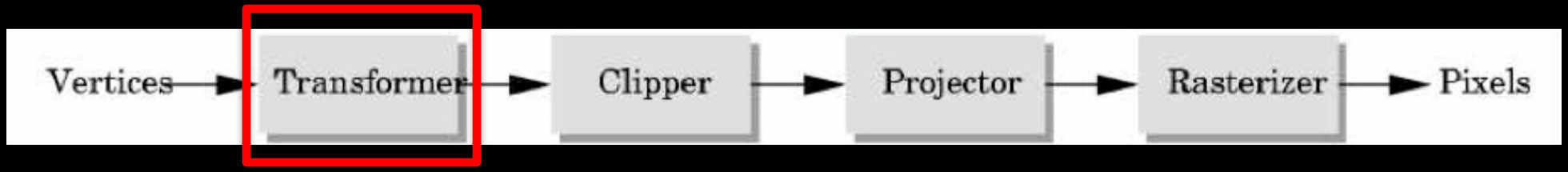
- Complex [Angel Ch. 3]

Transformer (core profile)



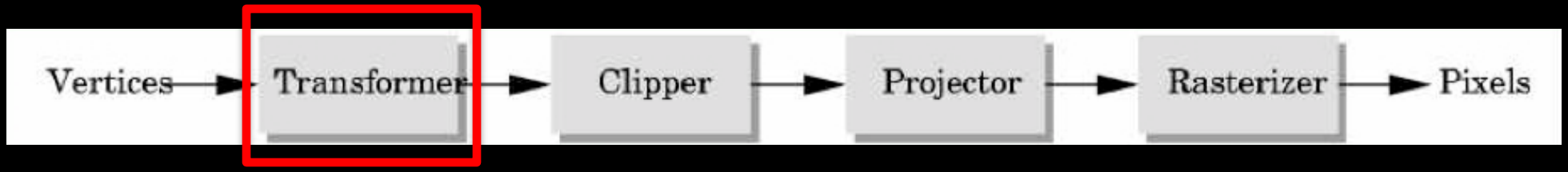
- Transformer in world coordinates
- 4x4 matrix
- Created manually by the user
- Transmitted to the shader program before rendering
- You write your own transformation with matrix multiplications, or by whatever other means you want!

Transformer (core profile)



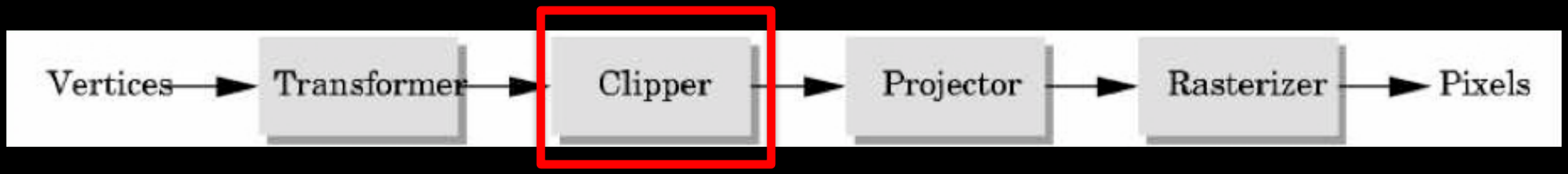
- 4x4 matrix
  - Why 4x4?
  - vertices are points in $\mathbb{R}^3$
  - Rotations are 3x3 matrices with determinant 1
  - So where does the 4th coordinate come from?

Transformer (core profile)

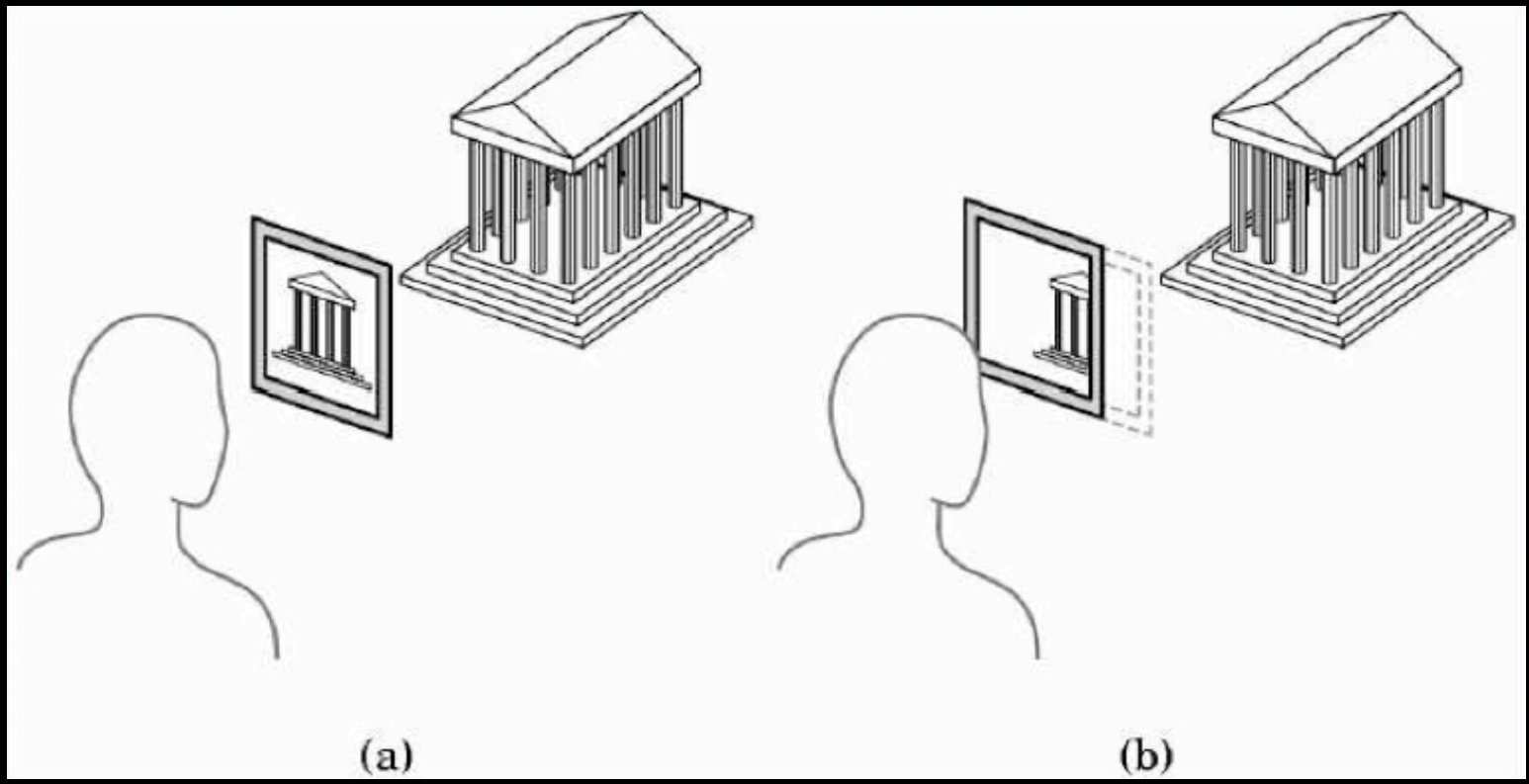


- 4x4 matrix
- **Homogeneous coordinates**
  - We can write $\mathbf{p} = (x, y, z)$, but then a matrix can only rotate: $\mathbf{p} \mapsto R\mathbf{p}$
  - If we instead write $\overline{\mathbf{p}} = (x, y, z, 1)$, then we can include translations:
  - $\overline{\mathbf{p}} \mapsto \begin{pmatrix} R & \mathbf{t} \\ 0 & 1 \end{pmatrix} \overline{\mathbf{p}} = R\mathbf{p} + \mathbf{t}$
  - (it's a bit more complicated)

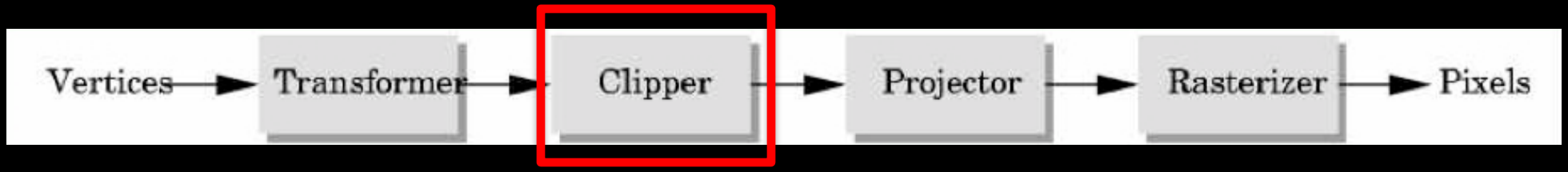
Clipper



- Mostly automatic (must set viewing volume)

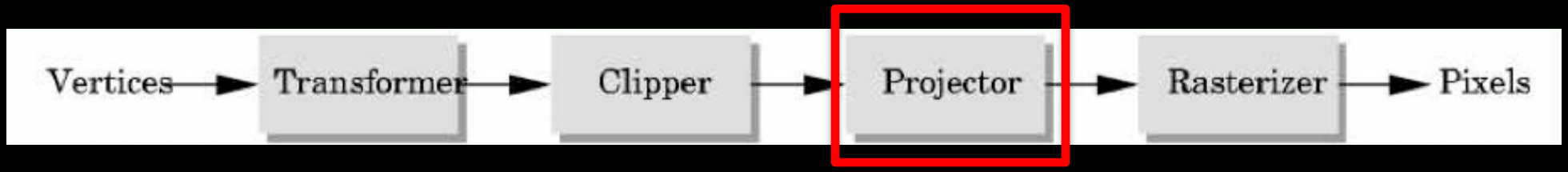


Clipper



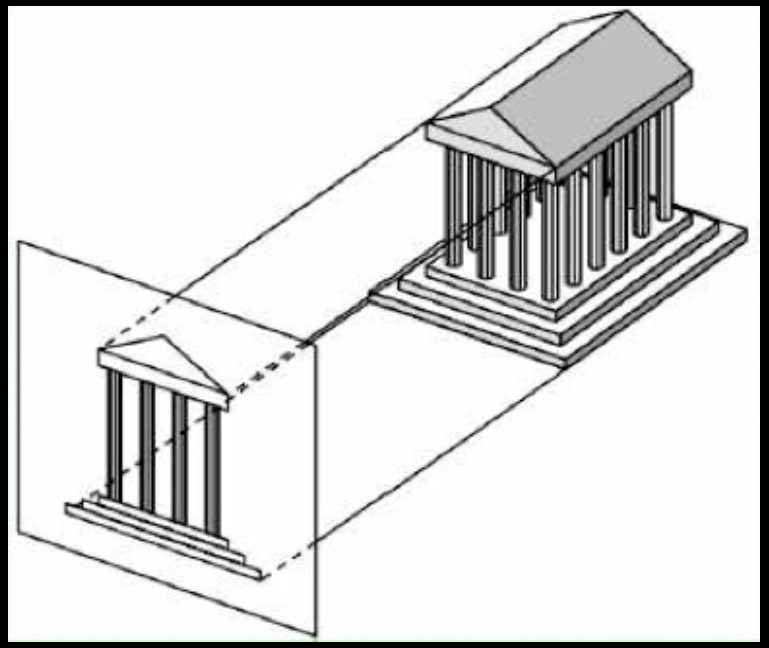
You do not fully control the clipper - OpenGL has a standard clipping setting. Make sure to transform into the clip space correctly!

Projector

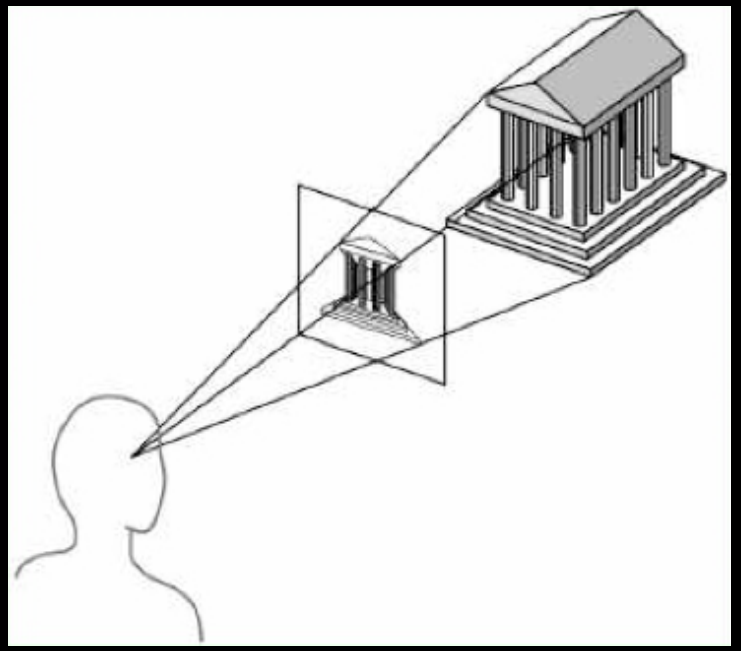


- Complex transformation [Angel Ch. 4]

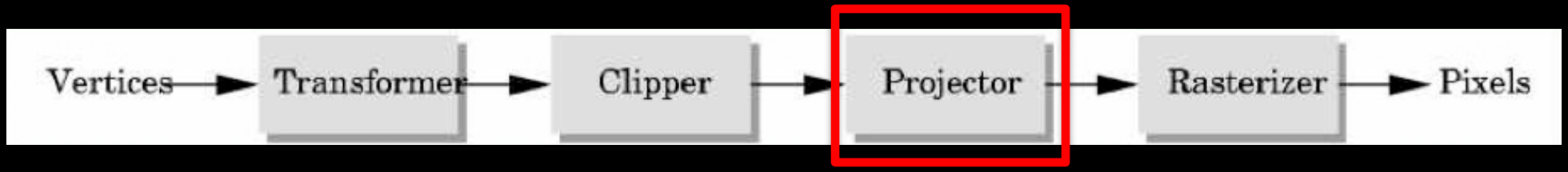
Orthographic



Perspective

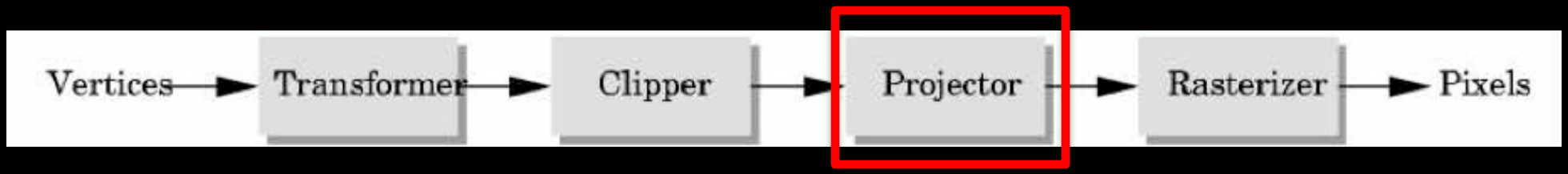


Projector



- In modern OpenGL, you have to do this yourself.
- This isn't hard, and there are good reasons to not hardcoding this.
- We will learn how to do these projections in the class.

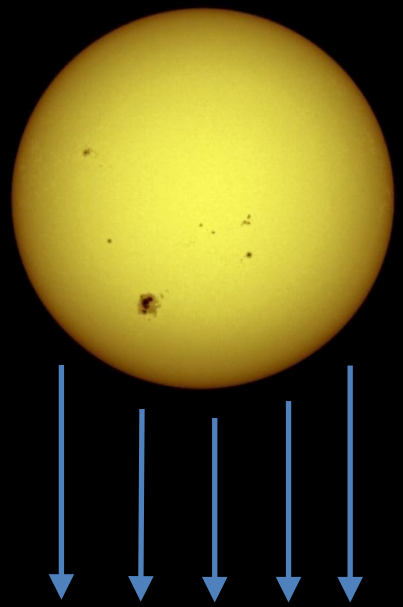
Projector



just kill the z coordinate

$(x, y, z) \mapsto (x, y)$

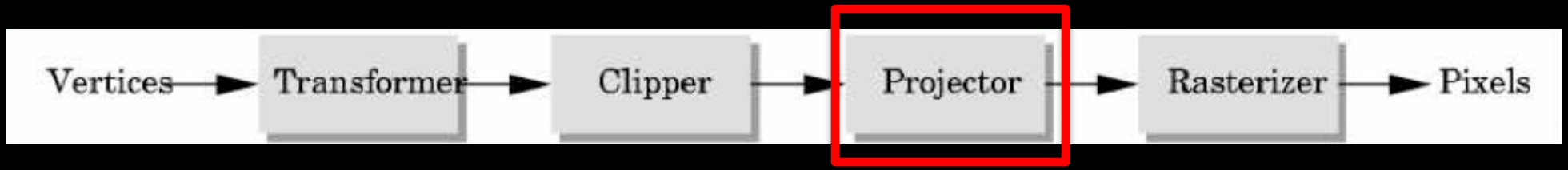
flattening

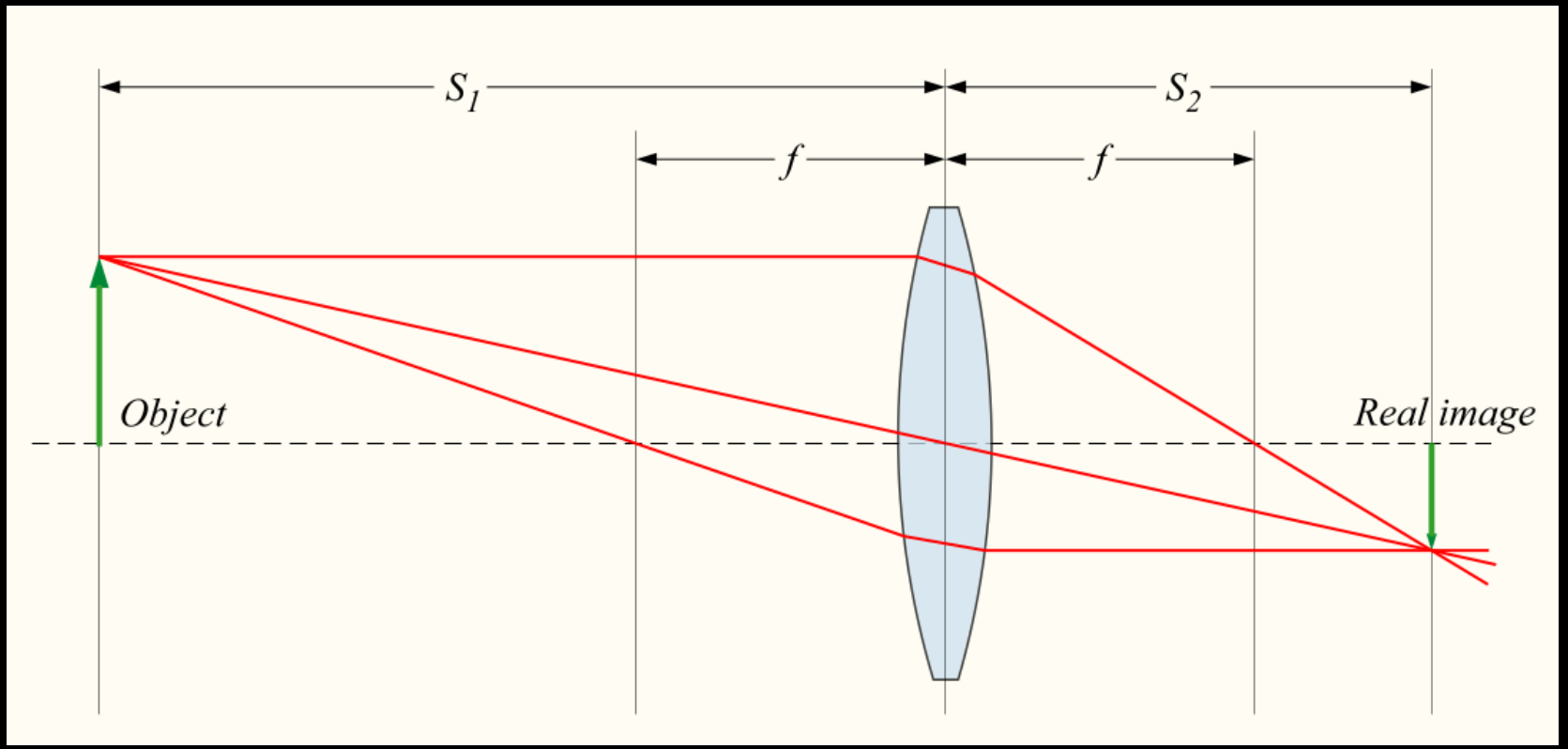


far away things

orthographic

Projector

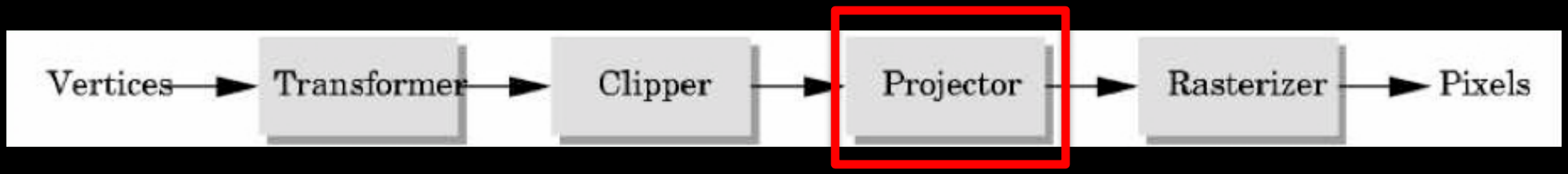


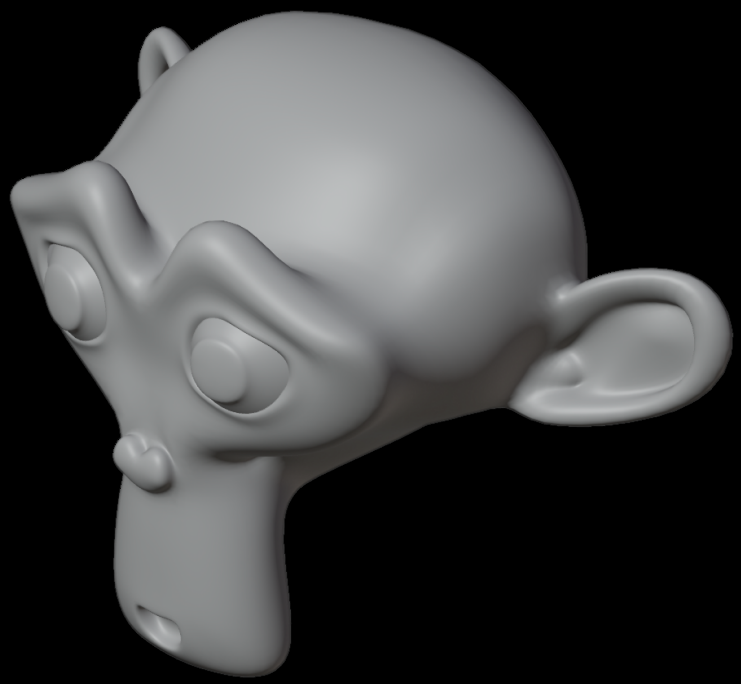


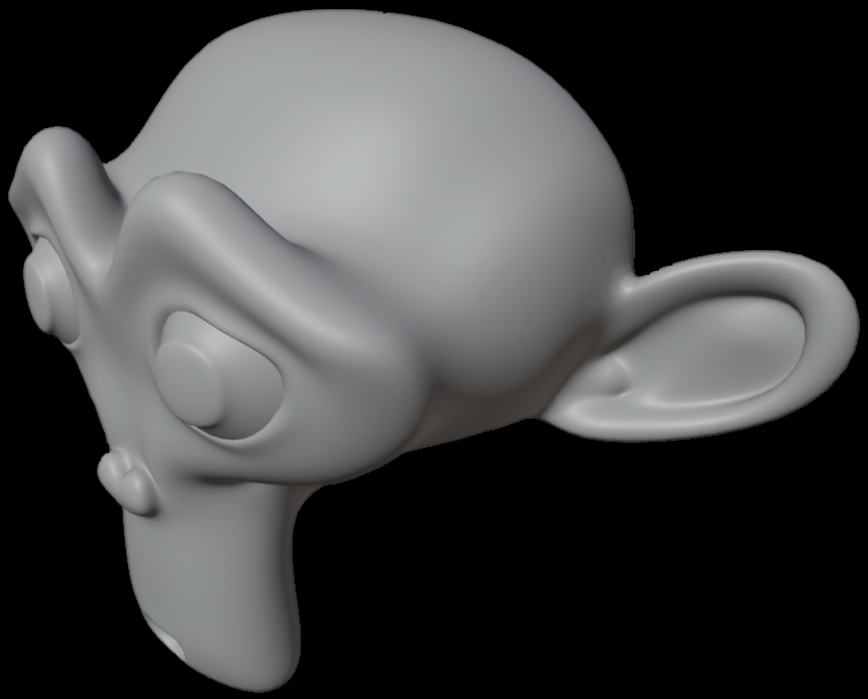
perspective

https://commons.wikimedia.org/wiki/File:Lens3.svg

Projector







orthographic

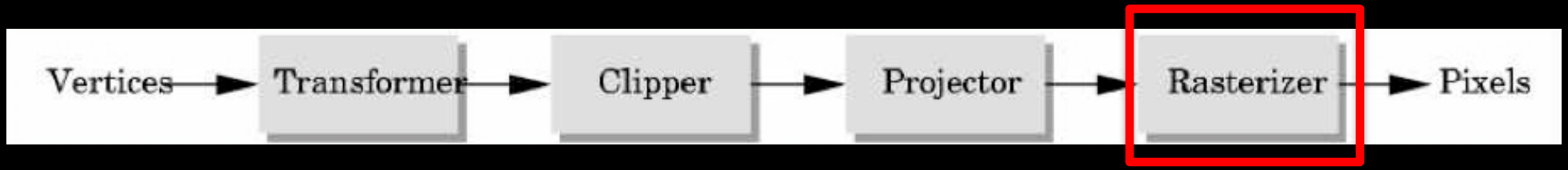
perspective

Projector

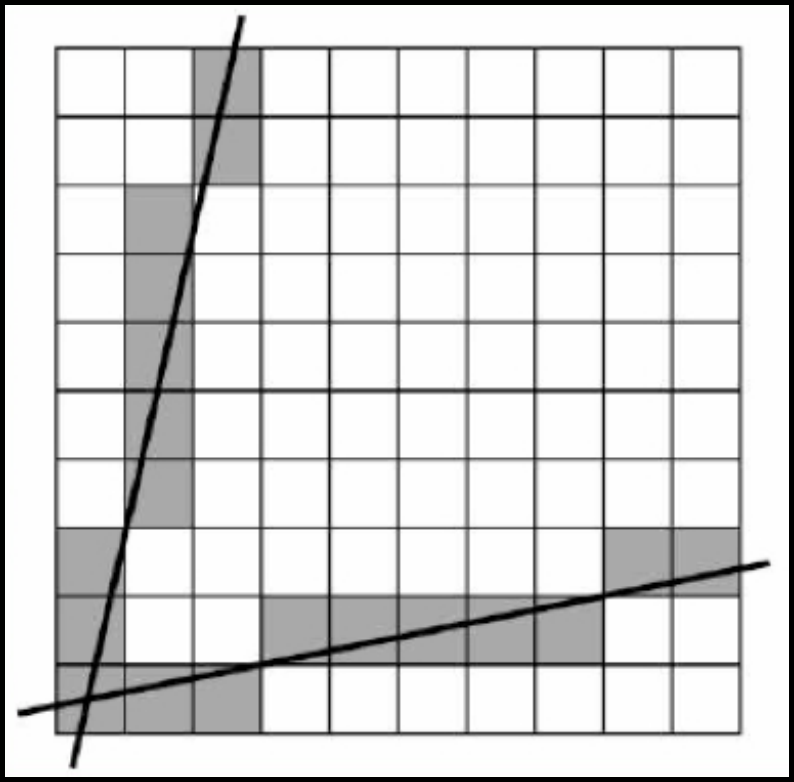
Dolly zoom

Jaws (1975)

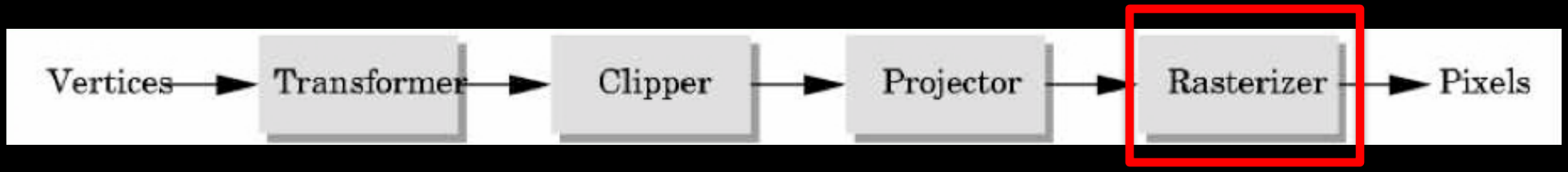
Rasterizer



- Interesting algorithms [Angel Ch. 6]
- To window coordinates
- Antialiasing

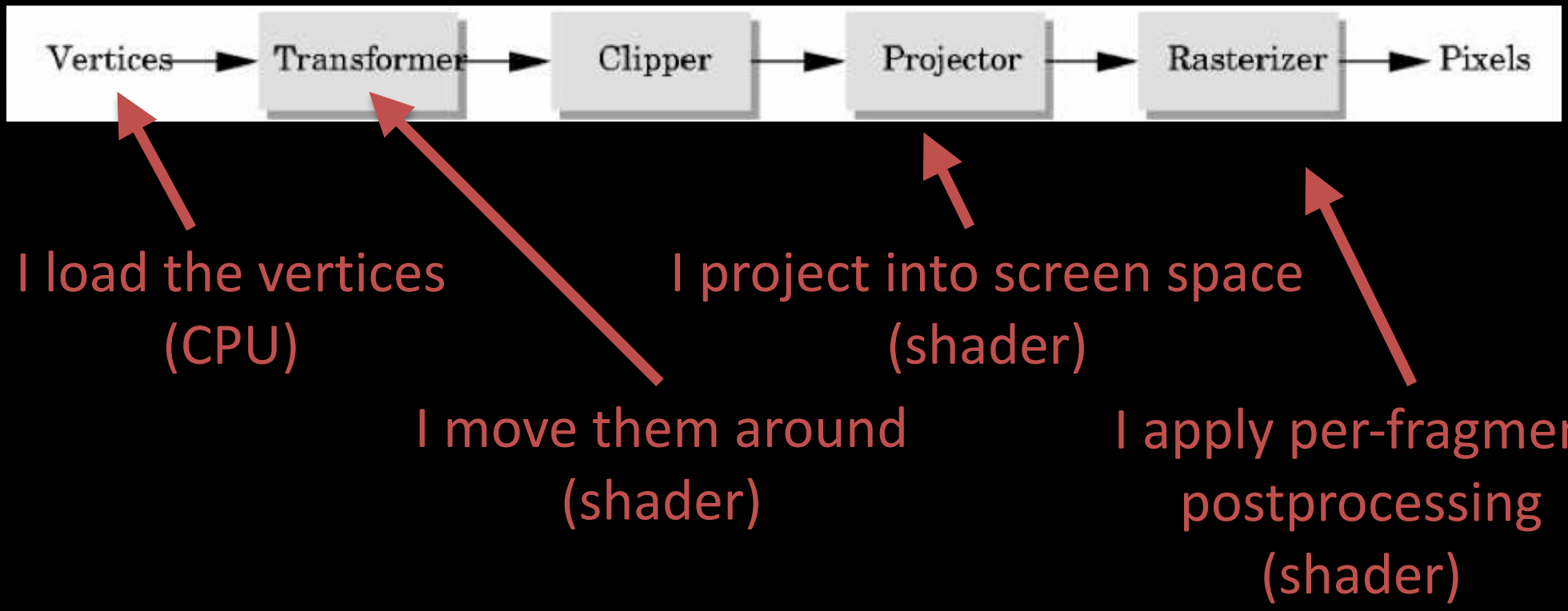


Rasterizer



- This is really, really difficult.
- More difficult than you think
- You do not need to deal with this - OpenGL will rasterize for you.

What do I influence here?

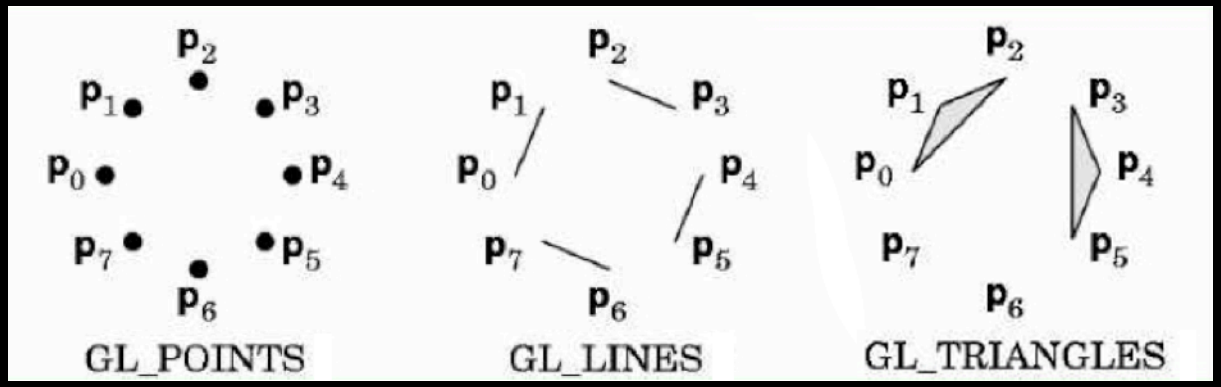


- (this is a simplification - there are a million things you can tweak here and there)
- (this ignores color and other per-vertex information that you can pass along in the pipeline)

Geometric Primitives

- Suppose we have 8 vertices:
  p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7

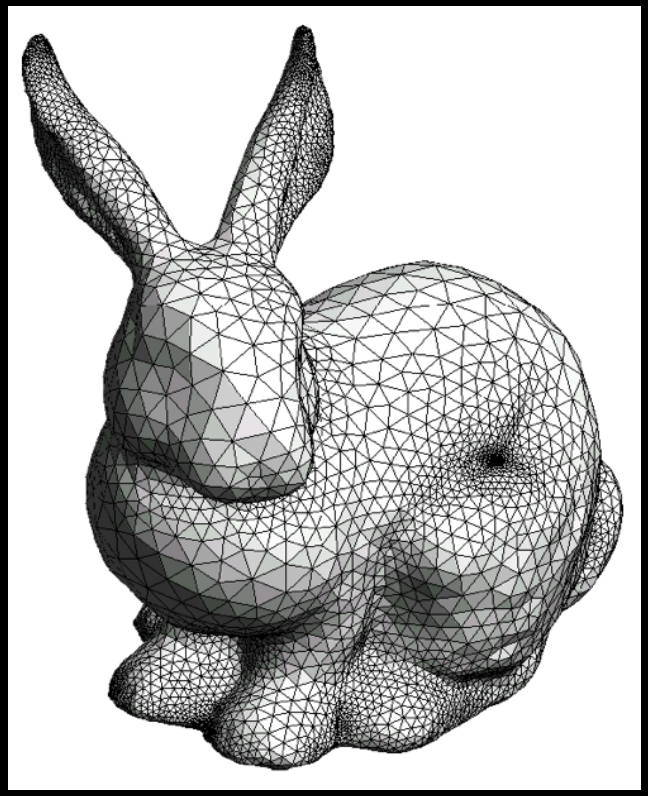
- Then, one can interpret them as:



- GL_POINTS, GL_LINES, GL_TRIANGLES are examples of primitive *type*

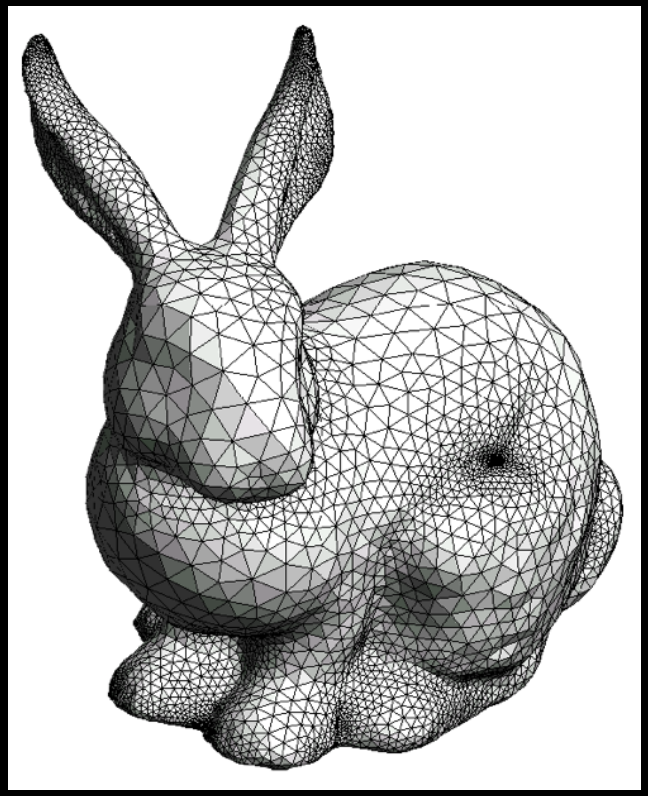
Triangles

- Can be any shape or size
- Well-shaped triangles have advantages for numerical simulation
- Shape quality makes little difference for basic OpenGL rendering



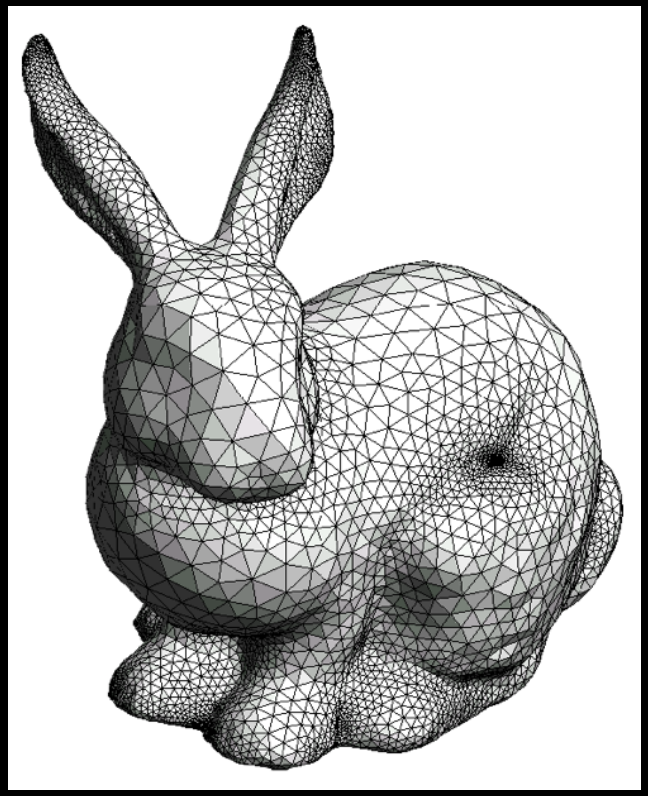
Triangles

- THIS IS YOUR MAIN PRIMITIVE
- Why would you use any other primitive?
- Many OpenGL implementations only reluctantly let you draw lines and points
  - On Mac, you can't set line thicknesses at all.

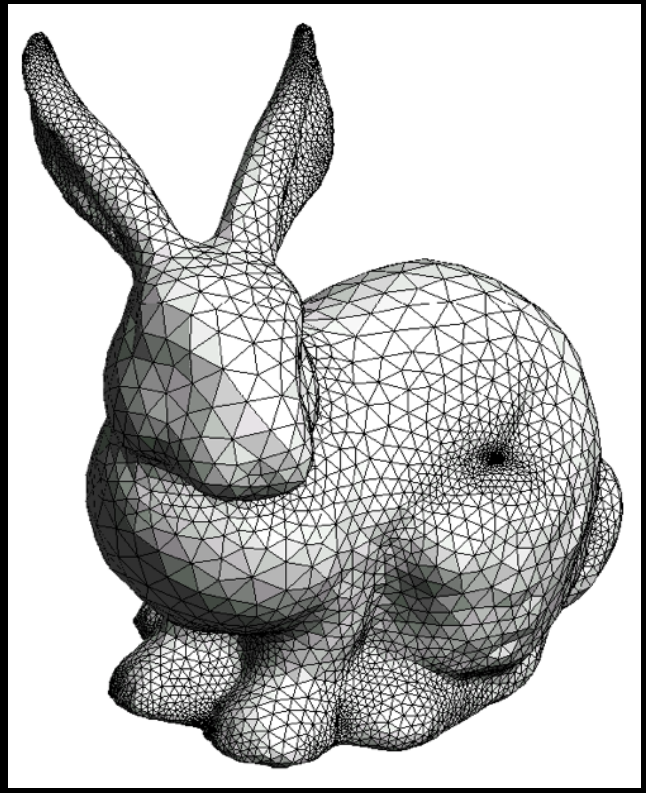


Triangles

- If you want to draw a line, or a point: fake it with triangles.
- If you want to draw a polygon: fake it with triangles.
- Modern GPUs only like to draw triangles.



Geometric Primitives (compatibility profile)



- Specified via vertices
- General schema

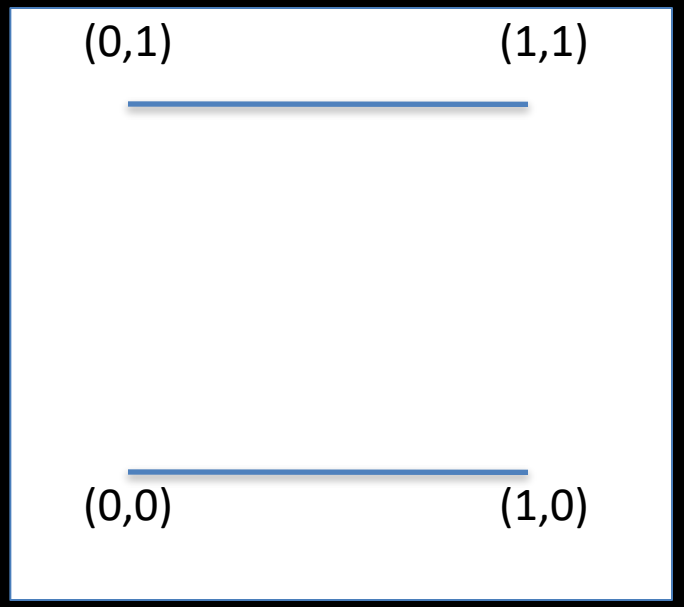
```
glBegin(type);
   glVertex3f(x1, y1, z1);
      ...
   glVertex3f(xN, yN, zN);
glEnd();
```

- *type* determines interpretation of vertices
- Can use glVertex2f(x,y) in 2D

Example: Draw Two Square Edges (compatibility profile)

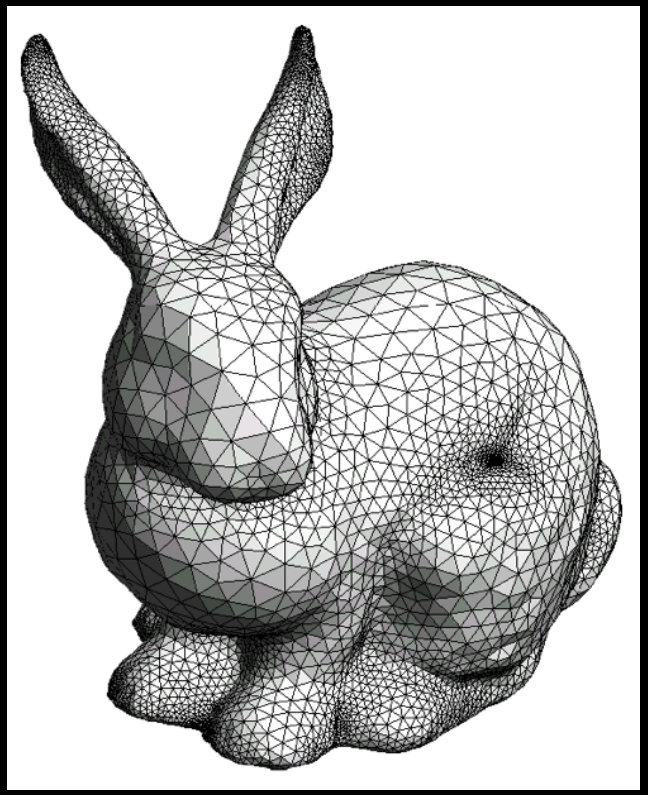
- *Type* = GL_LINES

```
glBegin(GL_LINES);
  glVertex3f(0.0, 0.0, -1.0);
  glVertex3f(1.0, 0.0, -1.0);
  glVertex3f(1.0, 1.0, -1.0);
  glVertex3f(0.0, 1.0, -1.0);
glEnd();
```



- Calls to other functions are allowed between glBegin(*type*) and glEnd();

Geometric Primitives (core profile)



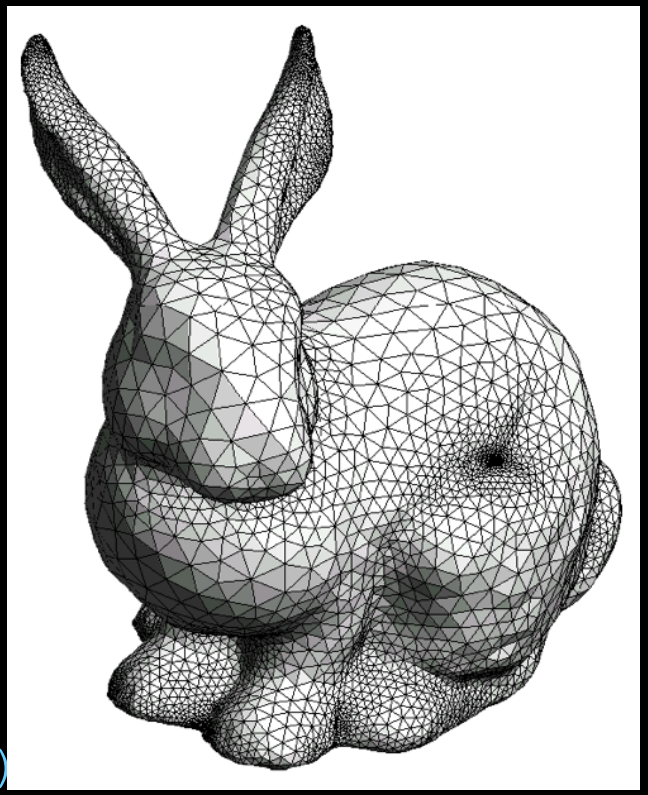
- Specified via vertices
- Stored in a Vertex Buffer Object (VBO)

```
int numVertices = 300;
float vertices[3 * numVertices];
// (… fill the "vertices" array …)
// create the VBO:
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
                              vertices, GL_STATIC_DRAW);
```

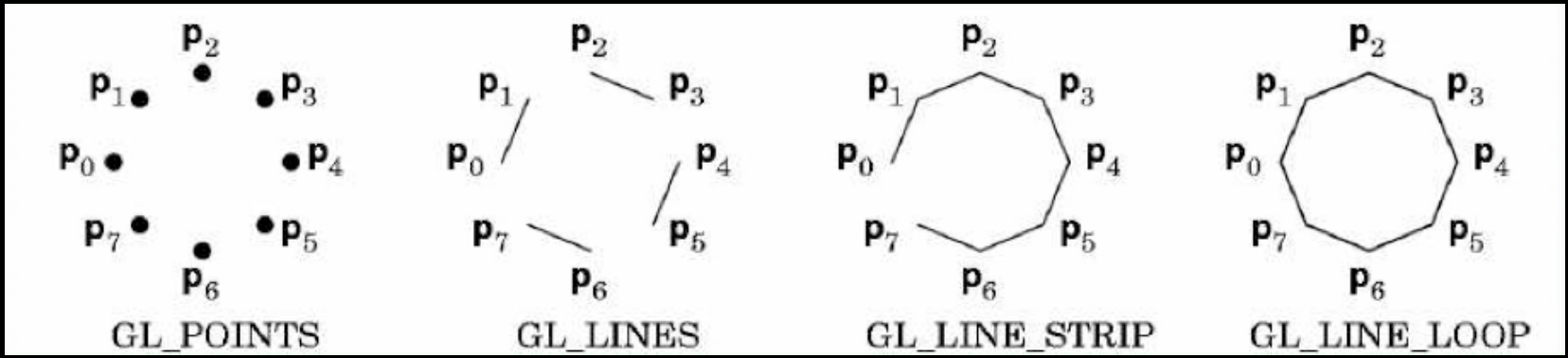
Clunky interface

- You have to be careful!
- Raw array
- No data structures
- You directly interact with memory

```
float vertices[3 * numVertices];
// (… fill the “vertices” array …)
```

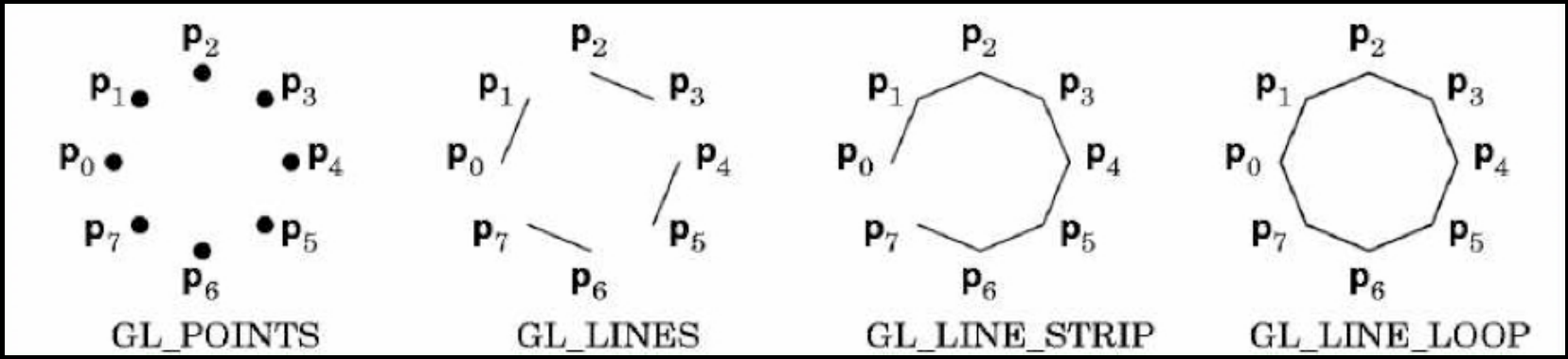


Render Points and Line Segments (compatibility profile)




```
glBegin (GL_POINTS);
// or GL_LINES to render lines
  glVertex3f(…);
   …
  glVertex3f(…);
glEnd();
```

Render Points and Line Segments (core profile)




```
// render points
glDrawArrays(GL_POINTS, 0, numVertices);
// render lines
glDrawArrays(GL_LINES, 0, numVertices);
```

Main difference between the two profiles

Compatibility:

Rendering:

```
glBegin(type);
  glVertex3f(x1, y1, z1);
  ...
  glVertex3f(xN, yN, zN);
glEnd();
```

Core:

Initialization:

```
int numVertices = 300;
float vertices[3 * numVertices];
// (… fill the "vertices" array …)
// create the VBO:
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vertices), vertices,
   GL_STATIC_DRAW);
```

Rendering:

```
glDrawArrays(type, 0, numVertices);
```

Common Bug

```
int numVertices = 50000;
float * vertices = (float*) malloc (sizeof(float) * 3 *
    numVertices);
…
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vertices), vertices, GL_STATIC_DRAW);
```

What is wrong?

(recall my clunkiness comment from earlier)

Common Bug

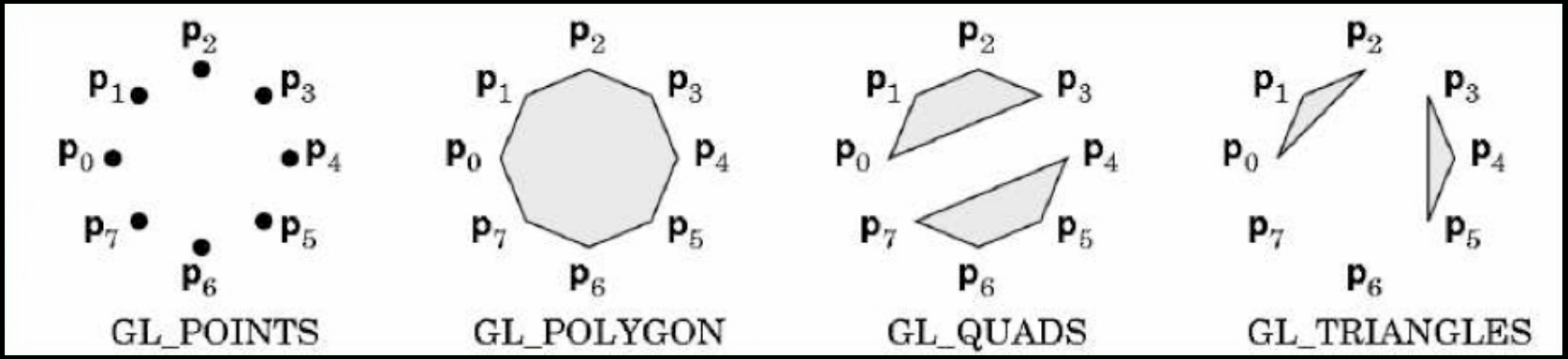
```
int numVertices = 50000;
float * vertices = (float*) malloc (sizeof(float) * 3
    * numVertices);
…
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vertices), vertices, GL_STATIC_DRAW);

glBufferData(GL_ARRAY_BUFFER,
   sizeof(float) * 3 * numVertices, vertices,
   GL_STATIC_DRAW);
```

Avoiding bugs

- Use C++.
- Write wrapper functions for these C functions that take C++ stl types like std::vector as input.
- Bugs are bad - use the tools of a modern programming language.
- Extensively test your interfaces and APIs.

Polygons

- Polygons enclose an area

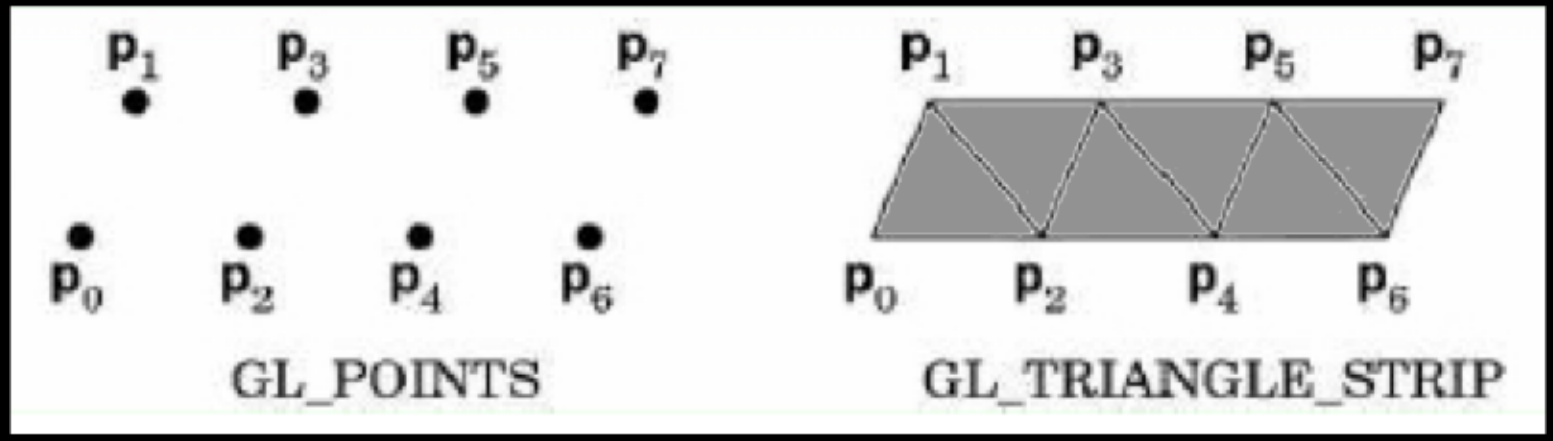


- Rendering of area (fill) depends on attributes
- All vertices must be in one plane in 3D
- GL_POLYGON and GL_QUADS are only available in the compatibility profile (removed in core profile since OpenGL 3.1)

(you should really only draw triangles)

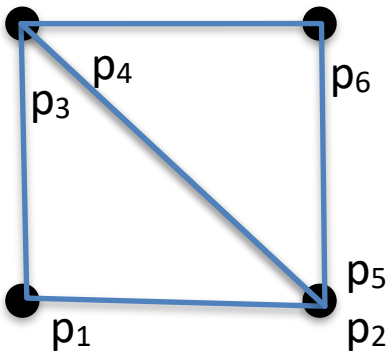
Triangle Strips

- Efficiency in space and time
- Reduces visual artifacts on old graphics cards

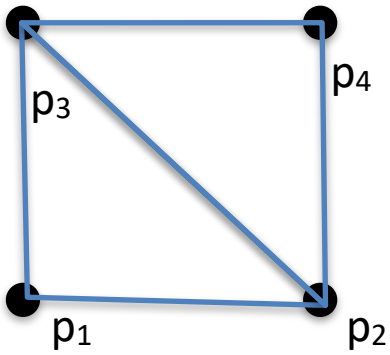


Triangle Meshes

- Don't use triangle strips
- If you want the space savings, use triangle meshes



individual triangles
stream of vertex triples
vertices = {p1, p2, p3, p4, p5, p6};



separate vertex list and index list
vertices = {p1, p2, p3, p4};
indices = {0,1,2,2,1,3};

- Makes certain shading techniques impossible.

Triangles in OpenGL Core ("modern")

Individual triangles:

Initialization:

```
int numTris = 50;
int numVertices = 3 * numTris;
float vertices[3 * numVertices];
// (… fill the "vertices" array …)
// (… 3 verts (9 floats) per tri …)
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vertices),
   vertices, GL_STATIC_DRAW);
```

Rendering:

```
glDrawArrays(GL_TRIANGLES, 0,
    numVertices);
```

Triangle mesh:

Initialization:

```
int numTris = 50;
int numVertices = ...;
float vertices[3 * numVertices];
// (… fill the "vertices" array …)
unsigned int indices[3 * numTris];
// (… fill the "indices" array, 3 uints per
tri …)
GLuint vbo, ebo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vertices), vertices,
   GL_STATIC_DRAW);
glGenBuffers(1, &ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
   sizeof(indices), indices,
   GL_STATIC_DRAW);
```

Rendering:

```
glDrawElements(GL_TRIANGLES, 3*numTris,
   GL_UNSIGNED_INT, 0);
```

Summary

1. Graphics pipeline
2. Primitives: vertices, lines, triangles