CS3451: Computer Graphics

GPU Rendering Pipeline

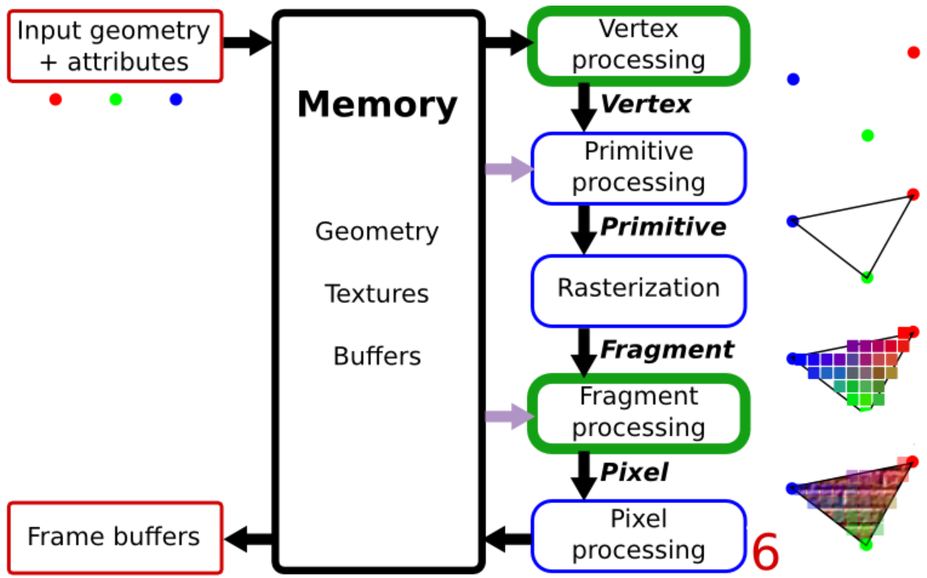
Bo Zhu

School of Interactive Computing

Georgia Institute of Technology

Study Plan

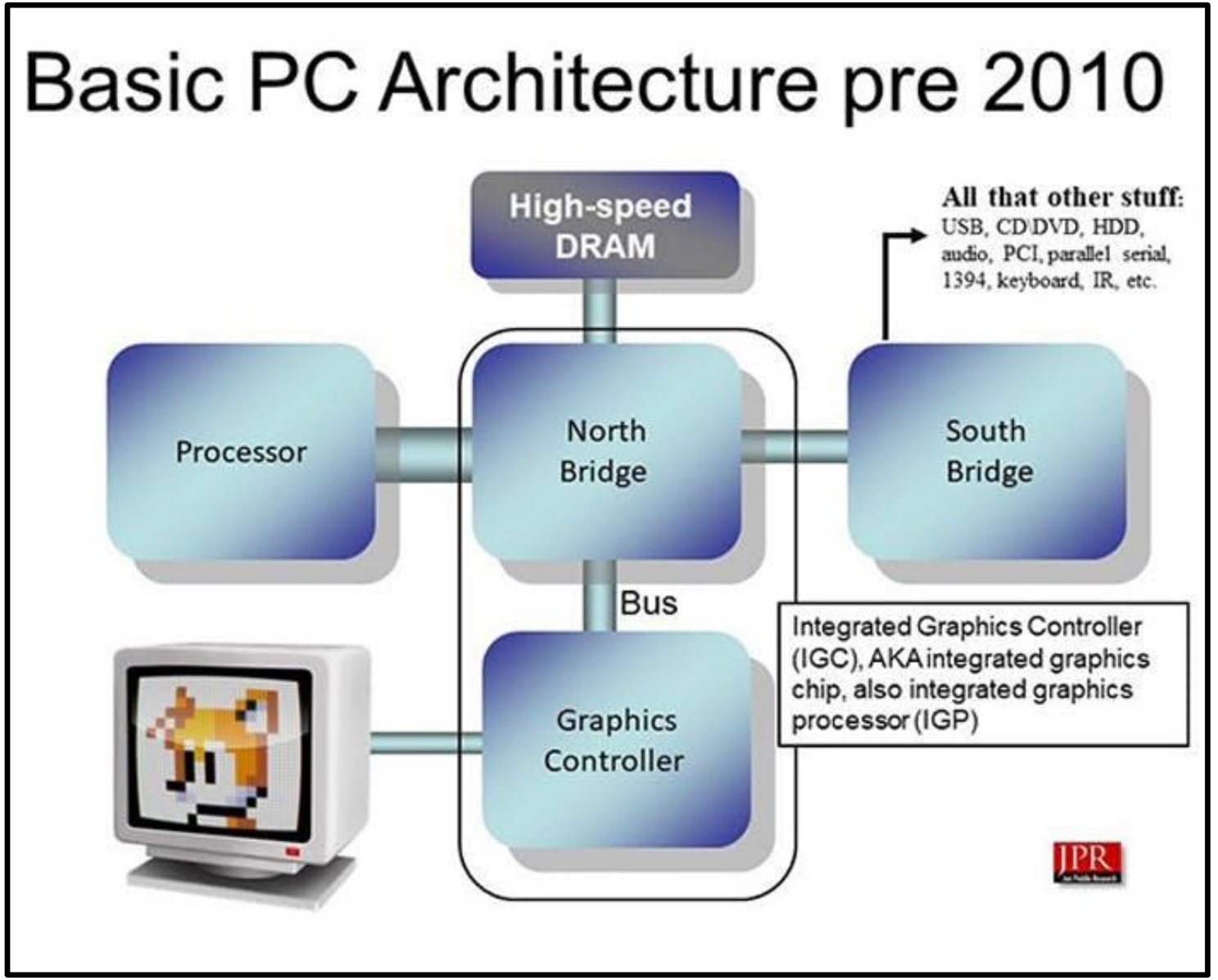
- PC Architecture and Graphics Processing
- GPU Programming Interfaces and OpenGL
- GPU Rendering Pipeline: Intuitive Explaination with Stanford Bunny
- GPU Rendering Pipeline: Technical Details
- CPU-GPU Data Communication



PC Architecture and Graphics Processing

Historical PC Architecture
with Integrated Graphics

- Integrated graphics controller embedded within the architecture
- Possesses basic graphics processing capabilities
- Lacks advanced lighting and shading effects due to limited computing power

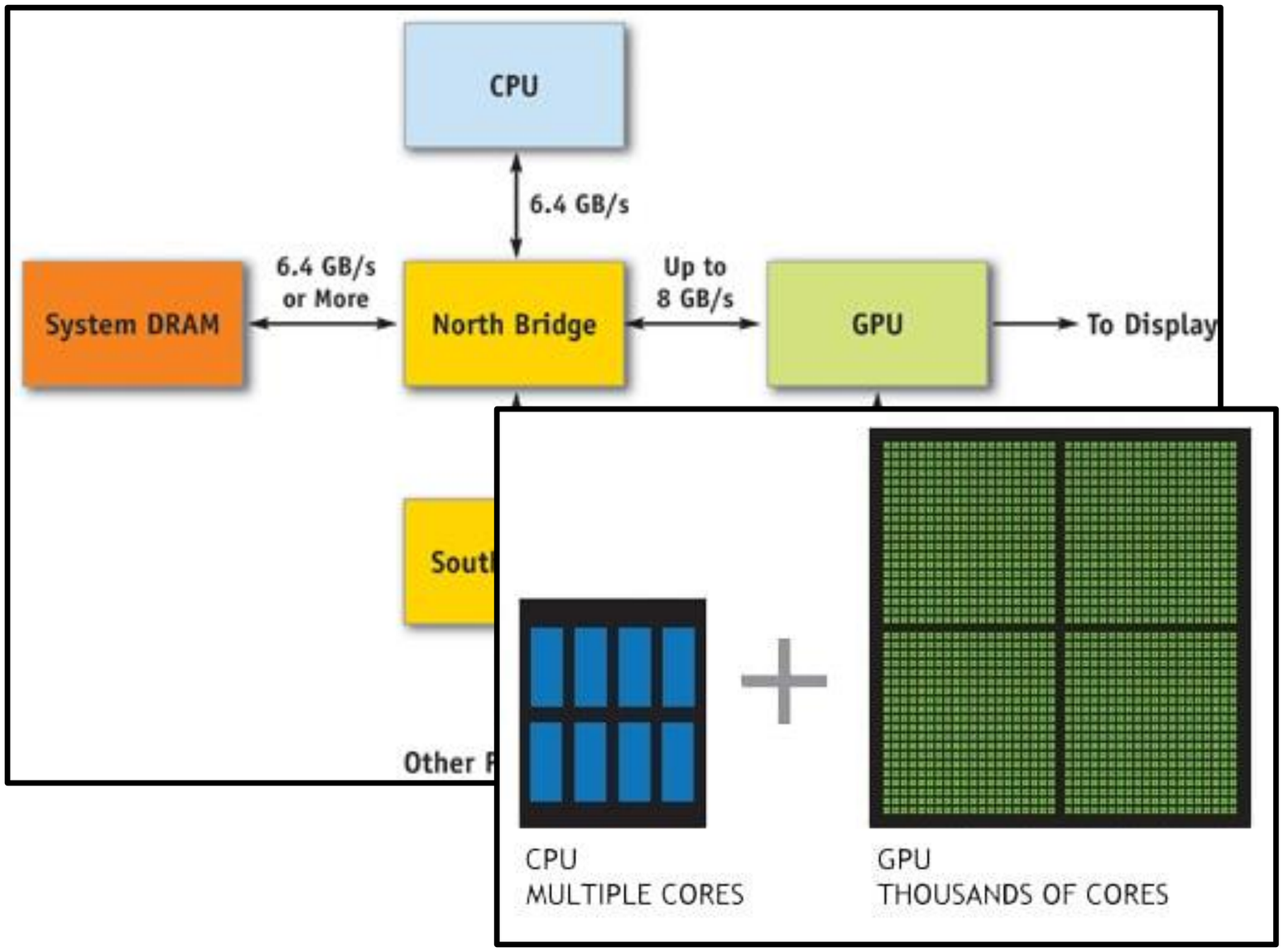


Our codebase supports integrated graphics cards ☺

Modern PC Architecture

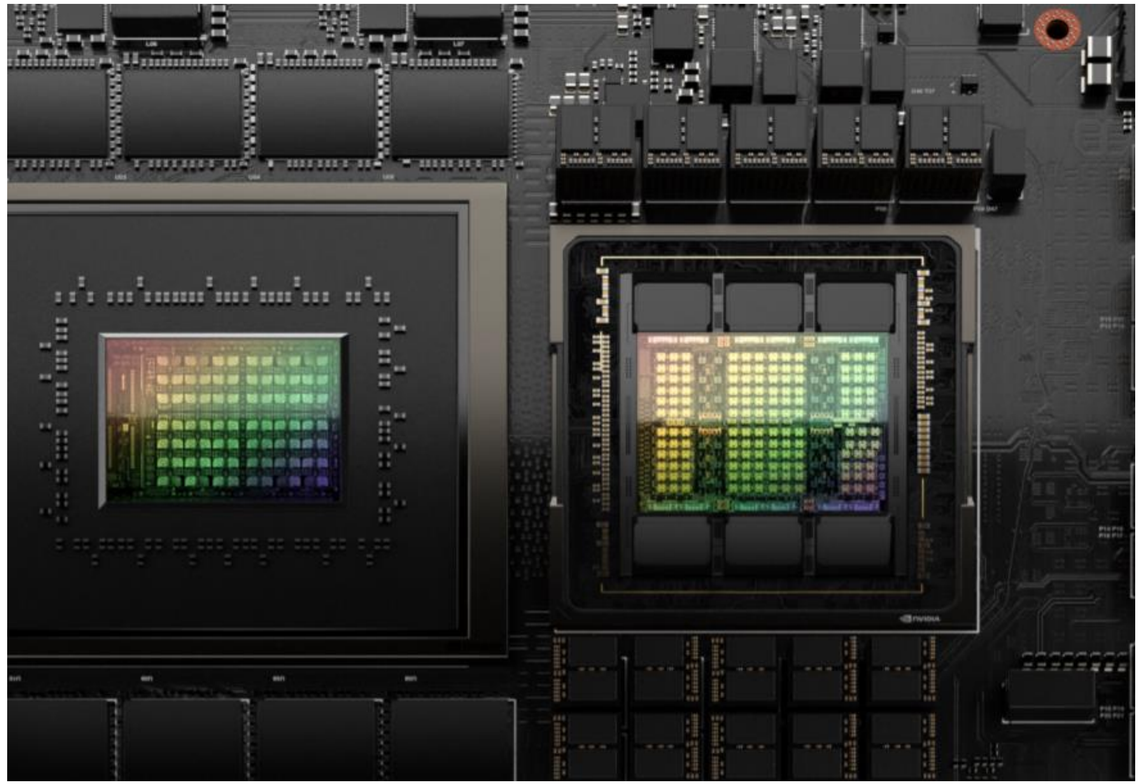
with independent GPU

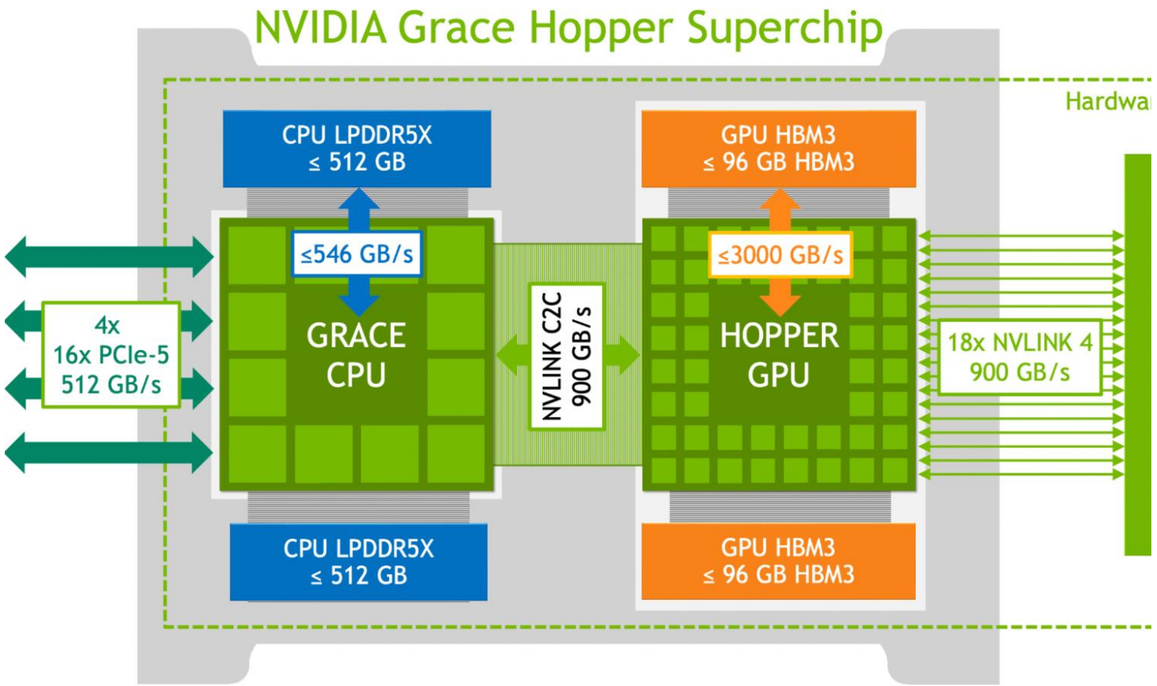
- CPU architected with independent GPU(s)
- A GPU has thousands of lightweighted processing cores
- High-speed data communication between CPU and GPU
- Possesses powerful **parallel** graphics processing capabilities



GPU-focused HPC Architecture

- Heterogeneous accelerated platform for high-performance computing (HPC) and AI workloads
- Accelerates applications with strength of both CPUs and GPUs

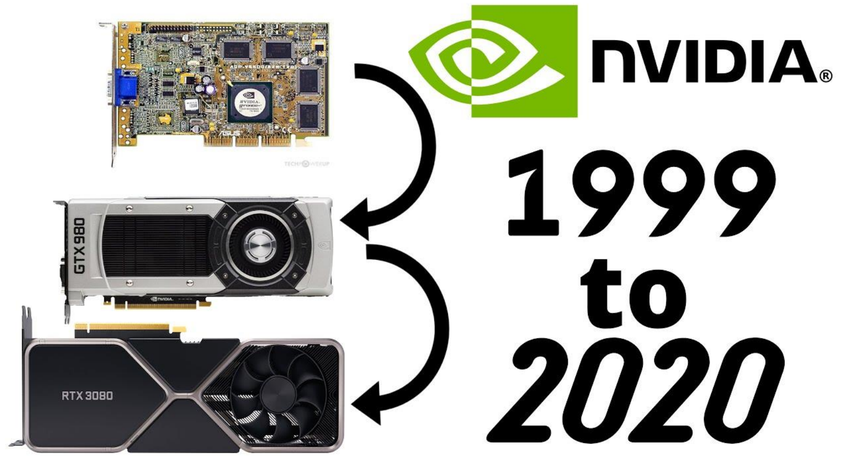




https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/

History of GPU Evolution

- GPU evolution history:
  - 1980's – No GPU. PC used integrated graphics controller
  - 1990's – Add more function into integrated graphics pipeline
  - 2000 – Beginning of independent GPU
  - 2005 – Massively parallel programmable processors
- Nowadays:
  - Highly parallel multiprocessor optimized for both graphic and general-purpose computing applications
  - New GPU are being developed every 12 to 18 months
  - Number crunching: 1 card ~= 1 teraflop ~= small cluster.



GPU Programming Interfaces and OpenGL

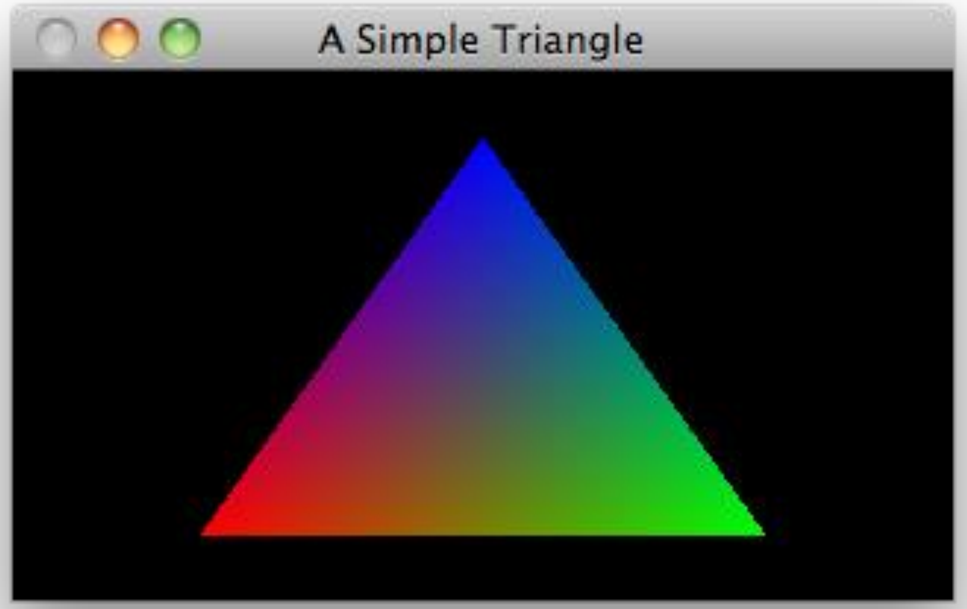
GPU Programming Interfaces

- **OpenGL** (Open Graphics Library) – an open standard for GPU graphics programming
  - Specifying the communication between CPU and GPU
  - Defining the programmable interface for graphics shaders
  - Industrial standard, accommodating a broad set of GPU types
  - The focus of this class
- **DirectX** – a series of Microsoft multimedia programming interfaces
- **CUDA** (Compute Unified Device Architecture) : parallel programming API created by NVIDA
  - Hardware and software architecture for issuing and managing computations on GPU
  - Massively parallel architecture. over 8000 threads is common
  - API libraries with C/C++/Fortran language
    - Numerical libraries: cuBLAS, cuFFT
- Others (**Vulkan**, **Metal**, **Mantle**, **WebGL**, etc.)

What Is OpenGL?

- OpenGL is a computer graphics rendering *application programming interface*, or API (for short)
  - With it, you can generate high-quality color images by rendering with geometric and image primitives
  - It forms the basis of many interactive applications that include 3D graphics
  - By using OpenGL, the graphics part of your application can be
    - operating system independent
    - window system independent

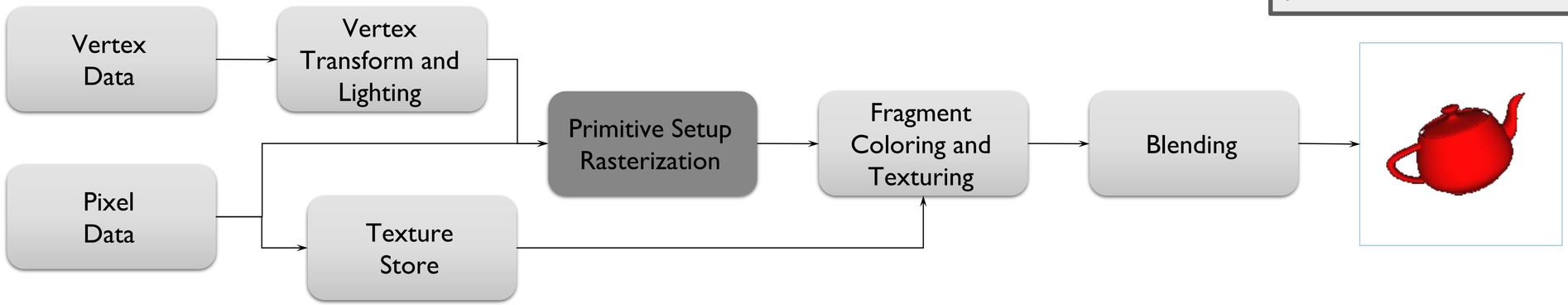
```
185     glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), (GLvoid*)(3 * sizeo
186     glBindBuffer(GL_ARRAY_BUFFER, 0);
187     glBindVertexArray(0);
188
189     // Load cube texture
190     GLuint textureID;
191     glGenTextures(1, &textureID);
192     int texWidth, texHeight;
193     unsigned char *image = SOIL_load_image(FileSystem::getPath("resources/textures/wood.png"
194
195     glBindTexture(GL_TEXTURE_2D, textureID);
196     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texWidth, texHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
197     glGenerateMipmap(GL_TEXTURE_2D);
198
199     glTexParameteri(GL_FRAMEBUFFER, GL_TEXTURE_WRAP_S, GL_REPEAT);
200     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
201     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
202     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
203     glBindTexture(GL_TEXTURE_2D, 0);
204     SOIL_free_image_data(image);
205
206     // Set up projection matrix
207     // TODO(Joey): check with new version of GLM and then use glm::radians
208     glm::mat4 projection = glm::perspective(45.0f, (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1
209     shader.Use(); // use before setting uniforms
210     glUniformMatrix4fv(glGetUniformLocation(shader.Program, "projection"), 1, GL_FALSE, glm:
211
212     glCheckError();
```



Fixed-Function Pipeline in the Beginning

```
void init(void)
{
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void display(void)
{
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_TRIANGLES);
glColor3f(1.0, 0.0, 0.0); glVertex3f(-1.0, -1.0, 0.0);
glColor3f(0.0, 1.0, 0.0); glVertex3f( 0.0,  1.0, 0.0);
glColor3f(0.0, 0.0, 1.0); glVertex3f( 1.0, -1.0, 0.0);
glEnd();
glutSwapBuffers();
}
void reshape(int w, int h)
{
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
}
```

- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation

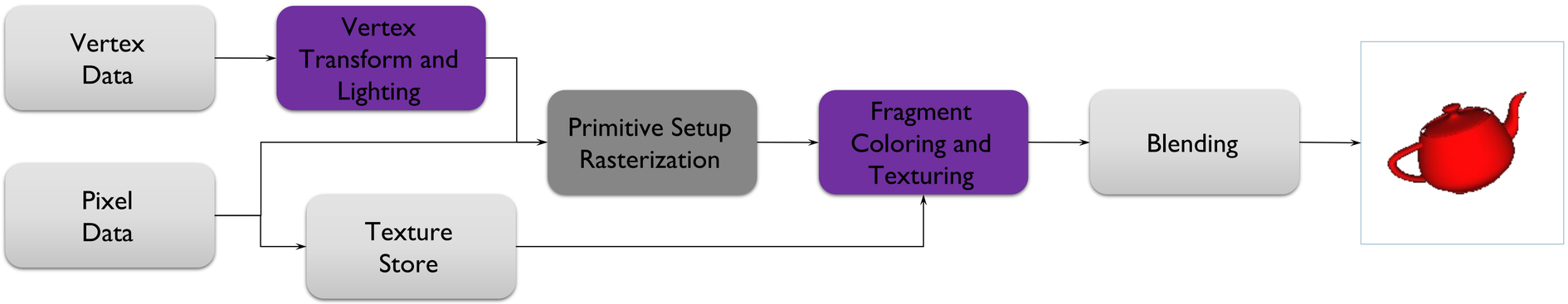


- The pipeline evolved
  - but remained based on fixed-function operation through OpenGL versions 1.1 through 2.0 (Sept. 2004)

Slides after SIGGAPH 2014 Course Notes

Programmable Pipeline

- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available



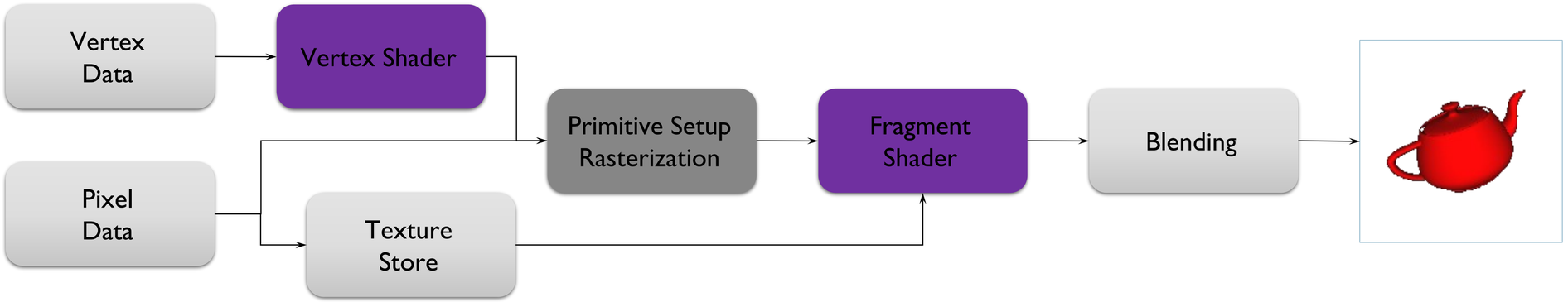
An Evolutionary Change

- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- Introduced a change in how OpenGL contexts are used

Context Type	Description
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

The Exclusively Programmable Pipeline

- OpenGL 3.1 removed the fixed-function pipeline
  - programs were required to use only shaders



- Additionally, almost all data is GPU-resident
  - all vertex data sent using buffer objects

Example: Data Transfer with Fixed Functions v.s. with Array Buffer

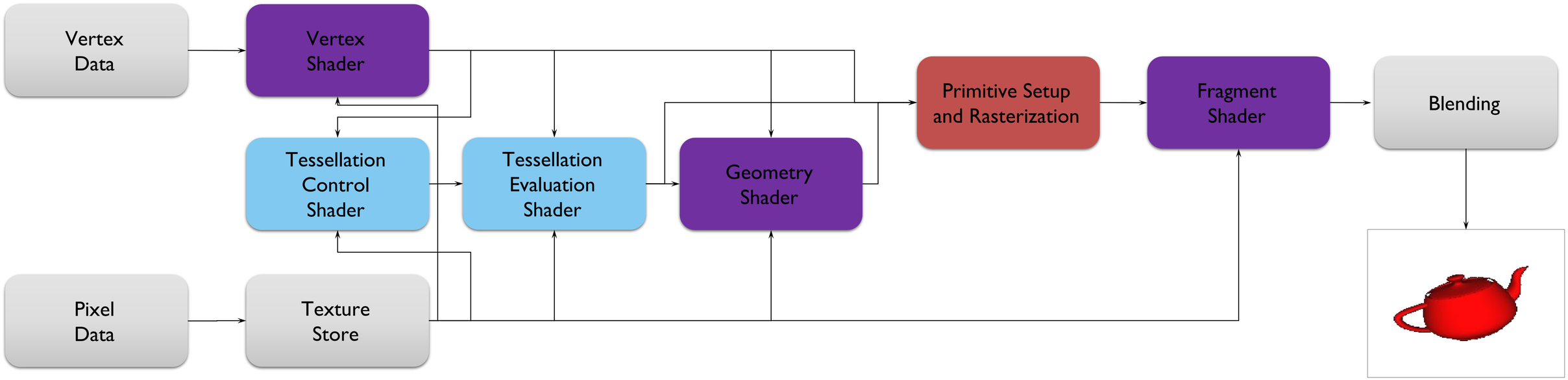
- Using arrays is much more flexible and efficient than fixed function calls!

```
void init(void)
{
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void display(void)
{
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_TRIANGLES);
glColor3f(1.0, 0.0, 0.0); glVertex3f(-1.0, -1.0, 0.0);
glColor3f(0.0, 1.0, 0.0); glVertex3f( 0.0,  1.0, 0.0);
glColor3f(0.0, 0.0, 1.0); glVertex3f( 1.0, -1.0, 0.0);
glEnd();
glutSwapBuffers();
}
void reshape(int w, int h)
{
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
}
```

```
void init(void)
{
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void display(void)
{
glClear(GL_COLOR_BUFFER_BIT);
static struct Vertex {
GLfloat position[3],
color[3];
} attribs[3] = {
{ {   -1.0, -1.0, 0.0 }, { 1.0, 0.0, 0.0 } },  // left-bottom,  red
{ {    0.0,  1.0, 0.0 }, { 0.0, 1.0, 0.0 } },  // center-top,   blue
{ {    1.0, -1.0, 0.0 }, { 0.0, 0.0, 1.0 } },  // right-bottom, green
};
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(Vertex), (char*)attribs + offsetof(Vertex, position));
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_FLOAT, sizeof(Vertex), (char*)attribs + offsetof(Vertex, color));
glDrawArrays (GL_TRIANGLES, 0, 3);
glutSwapBuffers();
}
void reshape(int w, int h)
{
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
}
```

The Latest Pipelines

- OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control* and *tessellation-evaluation* shaders
- Latest version is 4.6 in 2024



OpenGL ES and WebGL

- OpenGL ES 2.0
  - Designed for embedded and hand-held devices such as cell phones
  - Based on OpenGL 3.1
  - Shader based
- WebGL
  - JavaScript implementation of ES 2.0
  - Runs on most recent browsers

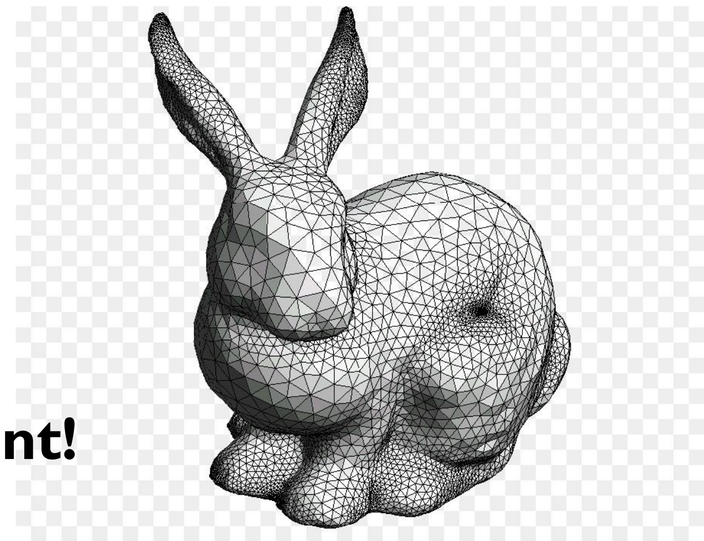
GPU Rendering Pipeline

How do we draw an object on computer screen?

- Using graphics hardware, via OpenGL, WebGL, DirectX, etc.
- GPUs do rasterization
  - The process of taking a triangle (or other primitive) and figuring out which pixels it covers

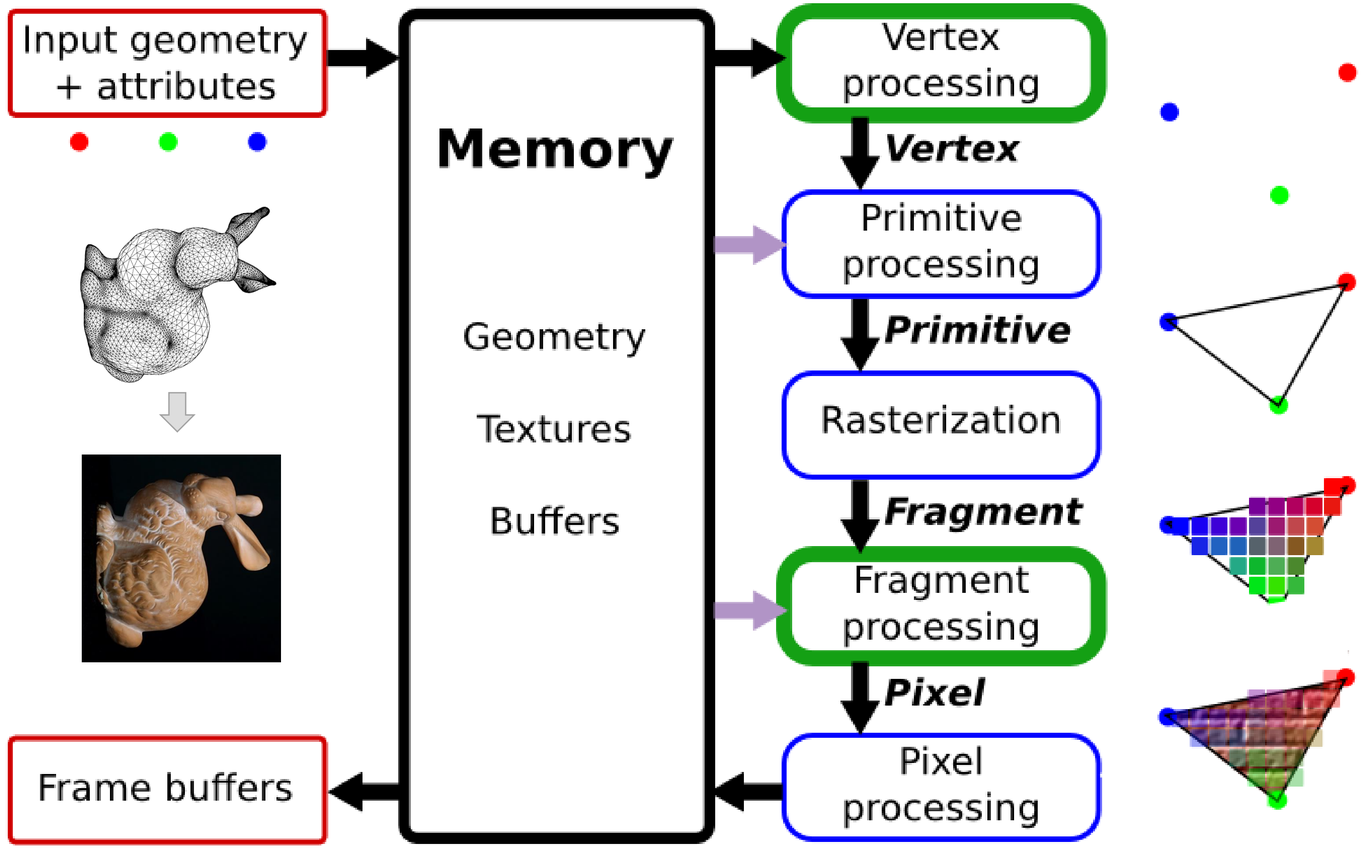
A conceptual algorithm:

```
for (each triangle)
   for (each pixel)
      if (triangle covers pixel)
         keep closest hit
```



Rendering for each triangle is independent!

Modern Graphics Pipeline



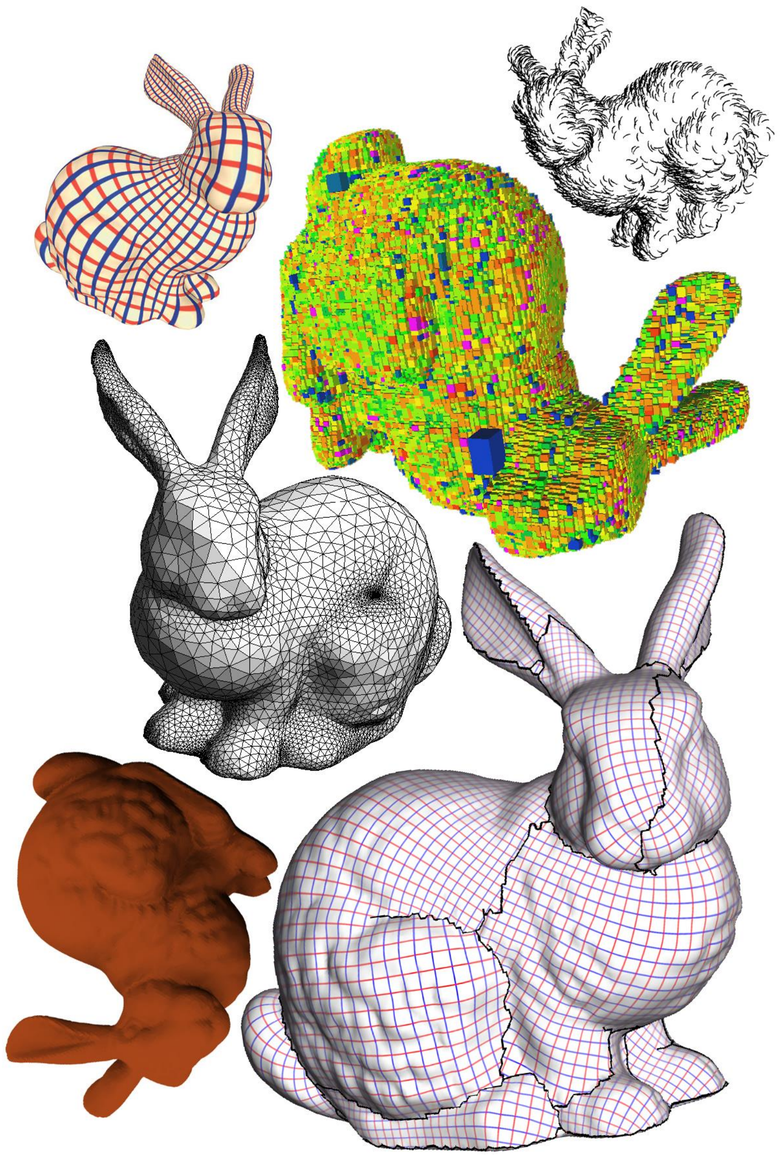
Important! Try to memorize every detail of this diagram!

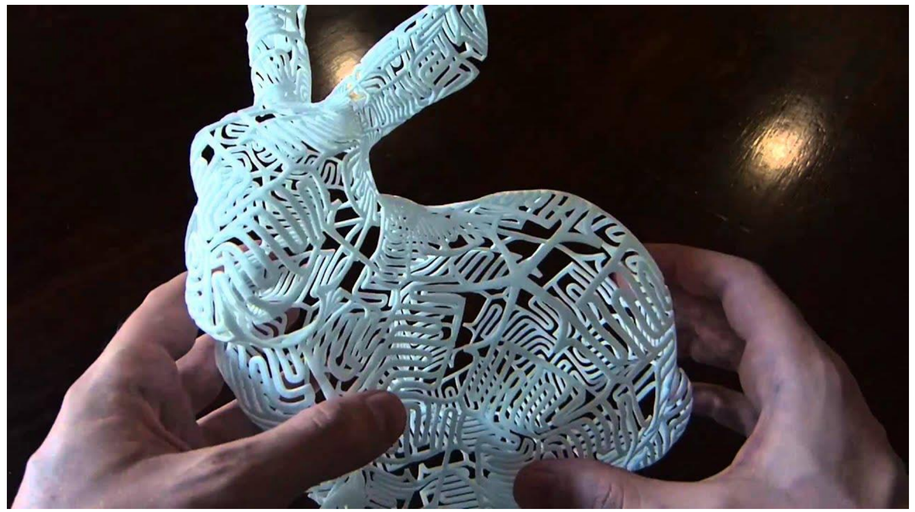
Let’s explain this with a cute bunny – Stanford Bunny

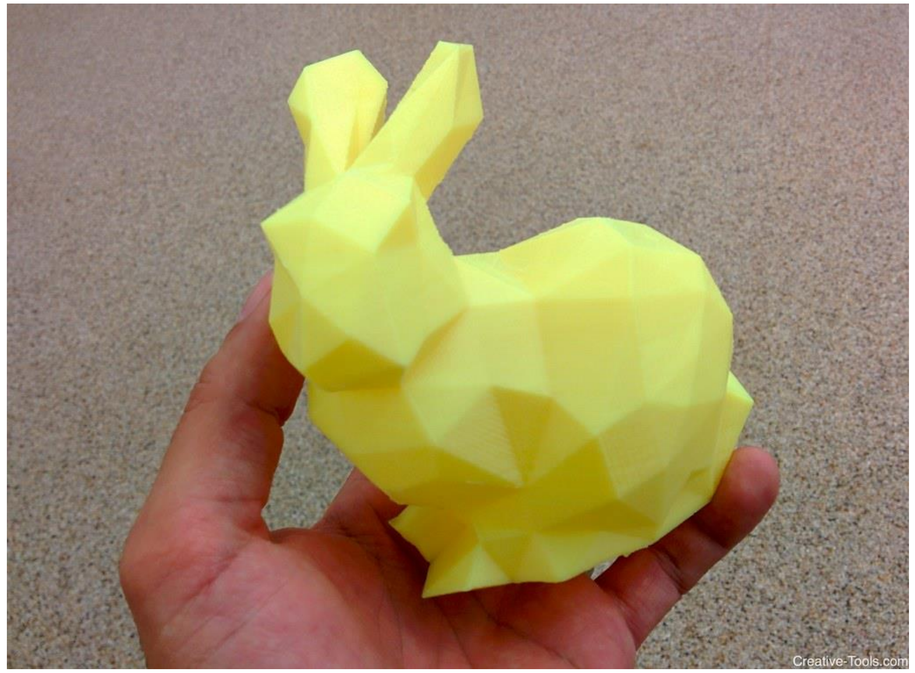
In the early 1990s, Stanford professor **Marc Levoy** and his postdoc, Greg Turk, created the world’s first seamless 3D computer model of a complex object using a range-finding laser scanner. Their object of choice, a terra cotta garden decoration, is now known worldwide as “**The Stanford Bunny**.” Computer graphics researchers still use it today. Approaching Easter, we talked to Levoy about what has kept his bunny going even three decades later.

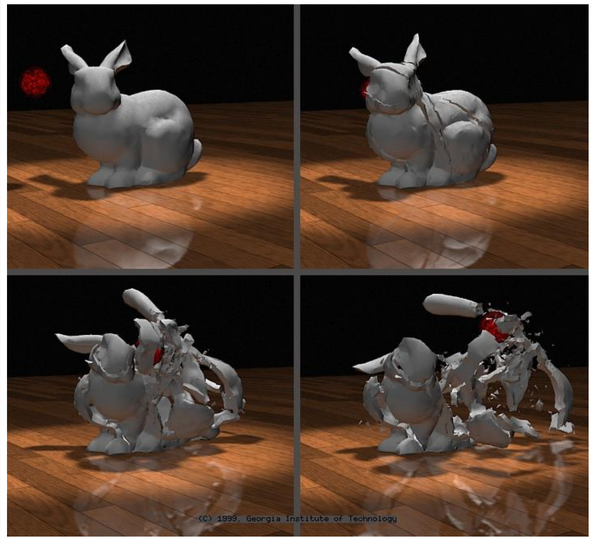
The tale of the ‘ubiquitous’ Stanford Bunny

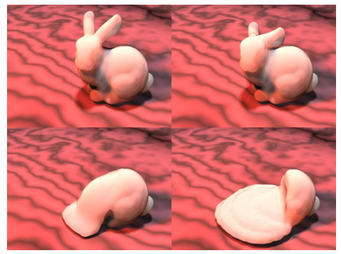
https://engineering.stanford.edu/magazine/tale-ubiquitous-stanford-bunny

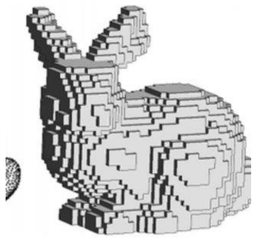








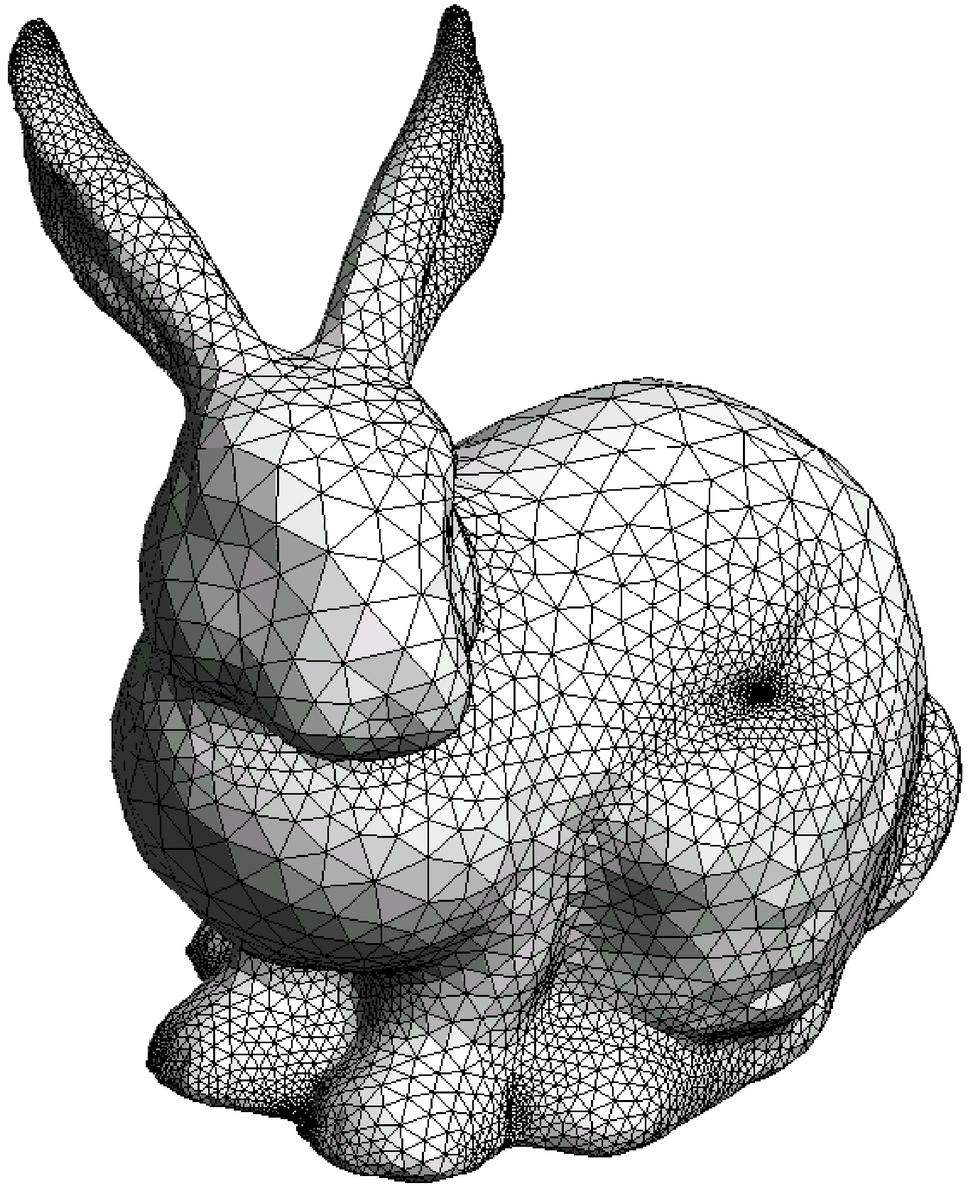




Stanford bunny is everywhere in graphics

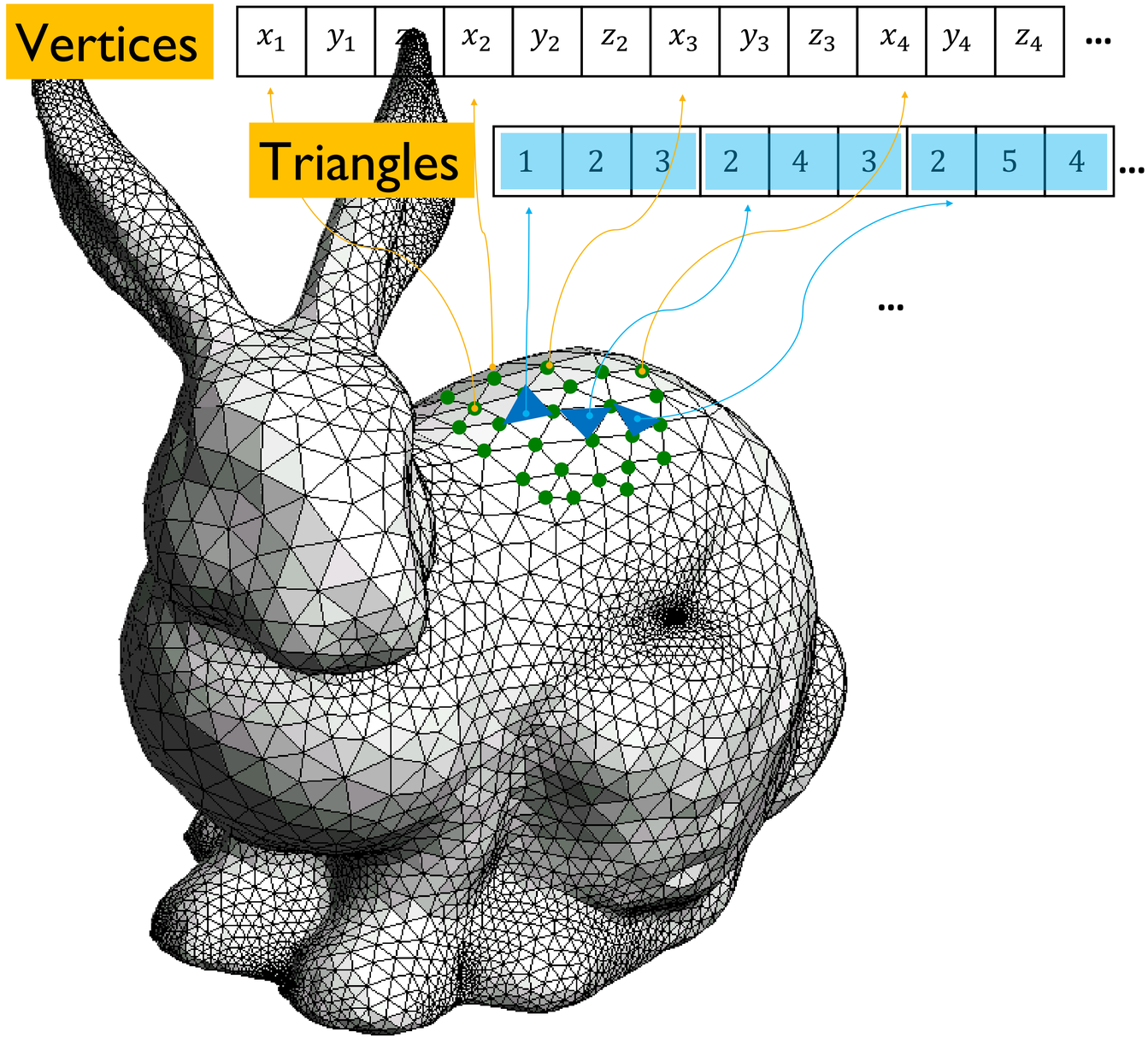
An intuitive explaination of graphics pipeline: display a Stanford bunny on screen

Represent the bunny using a triangle mesh on CPU



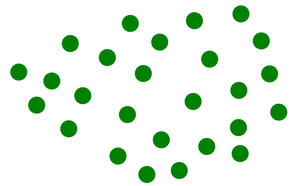
Mesh Data Load onto CPU

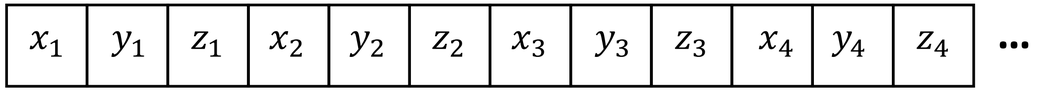
(I) Put all the vertices and triangles into two separate arrays, and send them to GPU

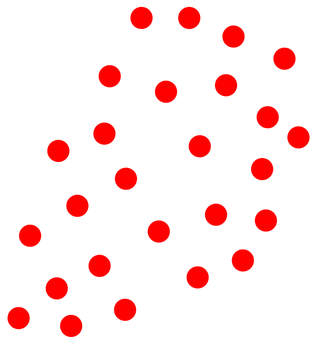


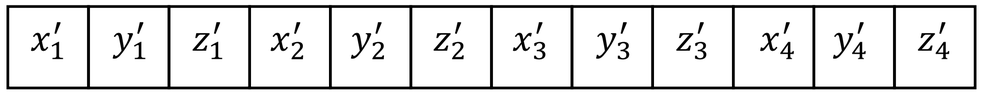
Data Transfer from CPU to GPU

(II) Employ transformation on vertices to obtain their new positions





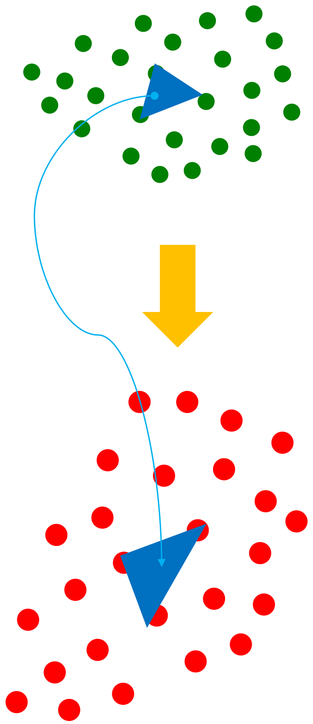


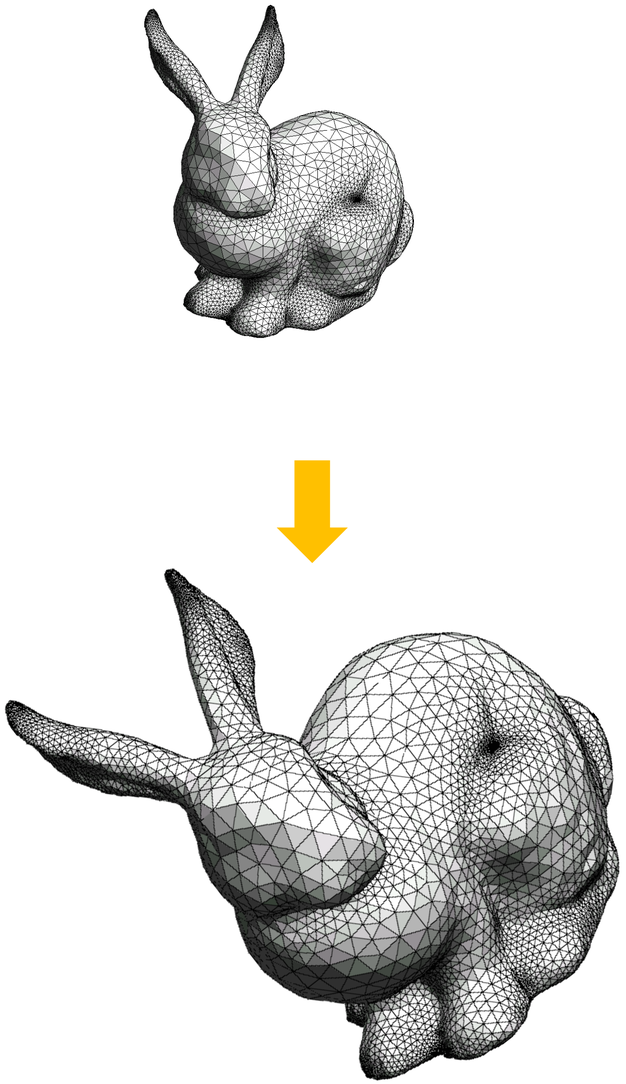


(e.g., scale, rotate, translate, etc.)

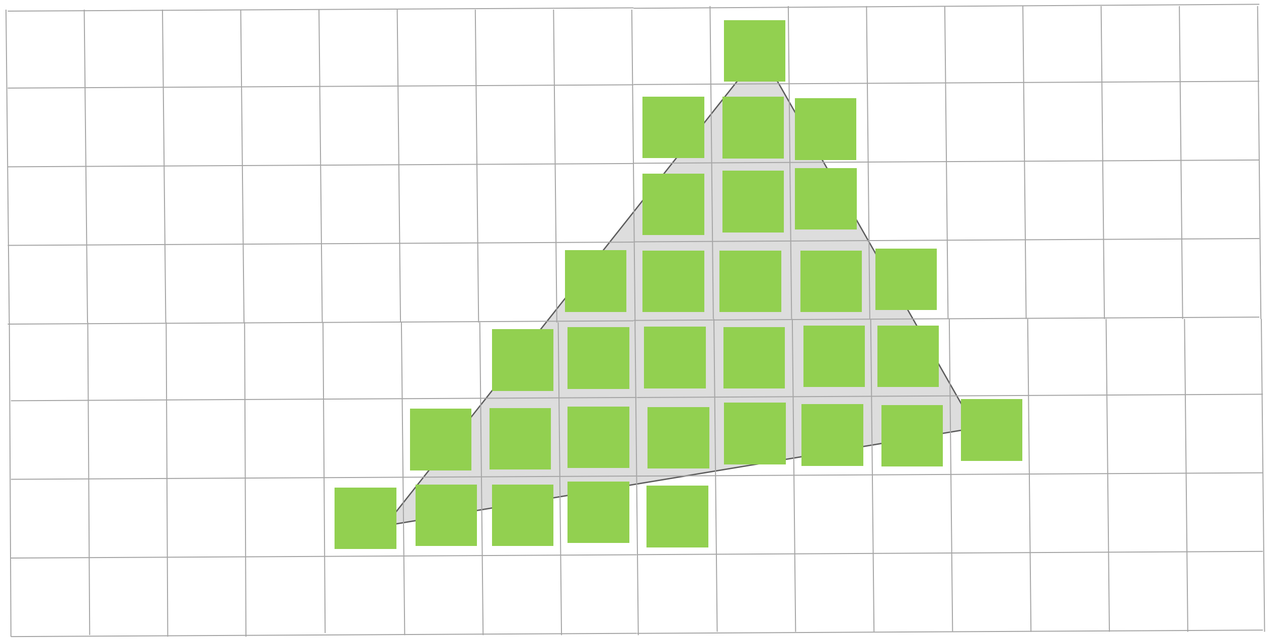
Vertex Processing on GPU

(III) All the triangles will be transformed according to the new vertex positions

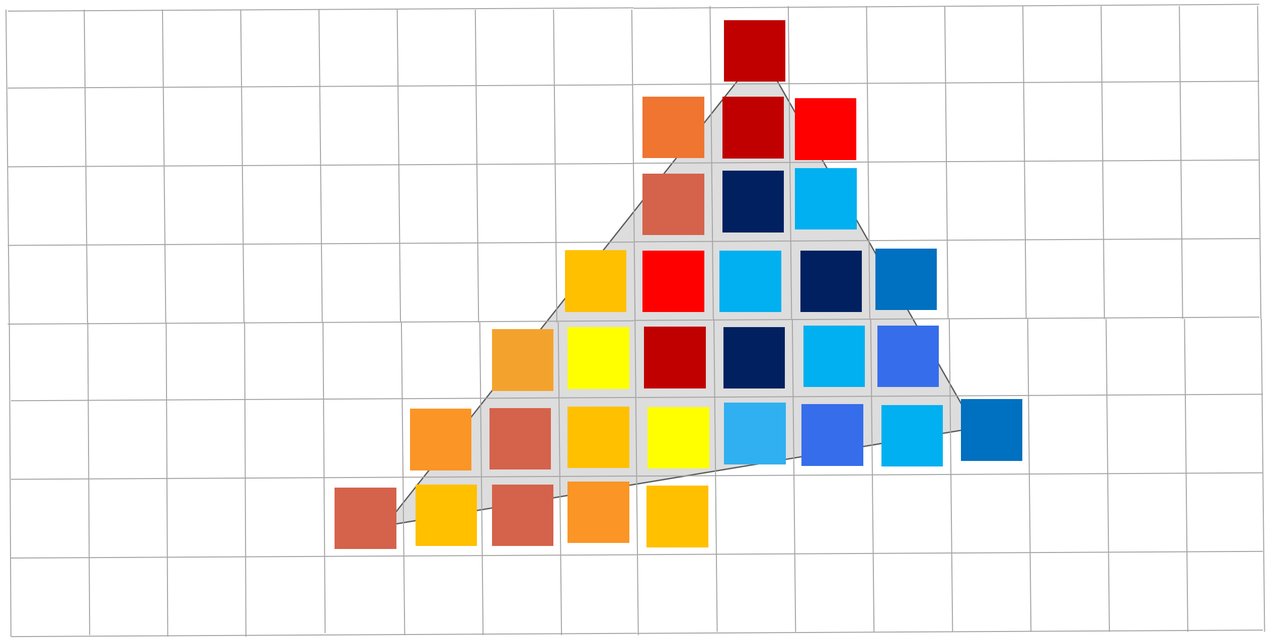




(IV) For each triangle, divide its shape into many small pieces on the screen



(V) Paint color on these small pieces

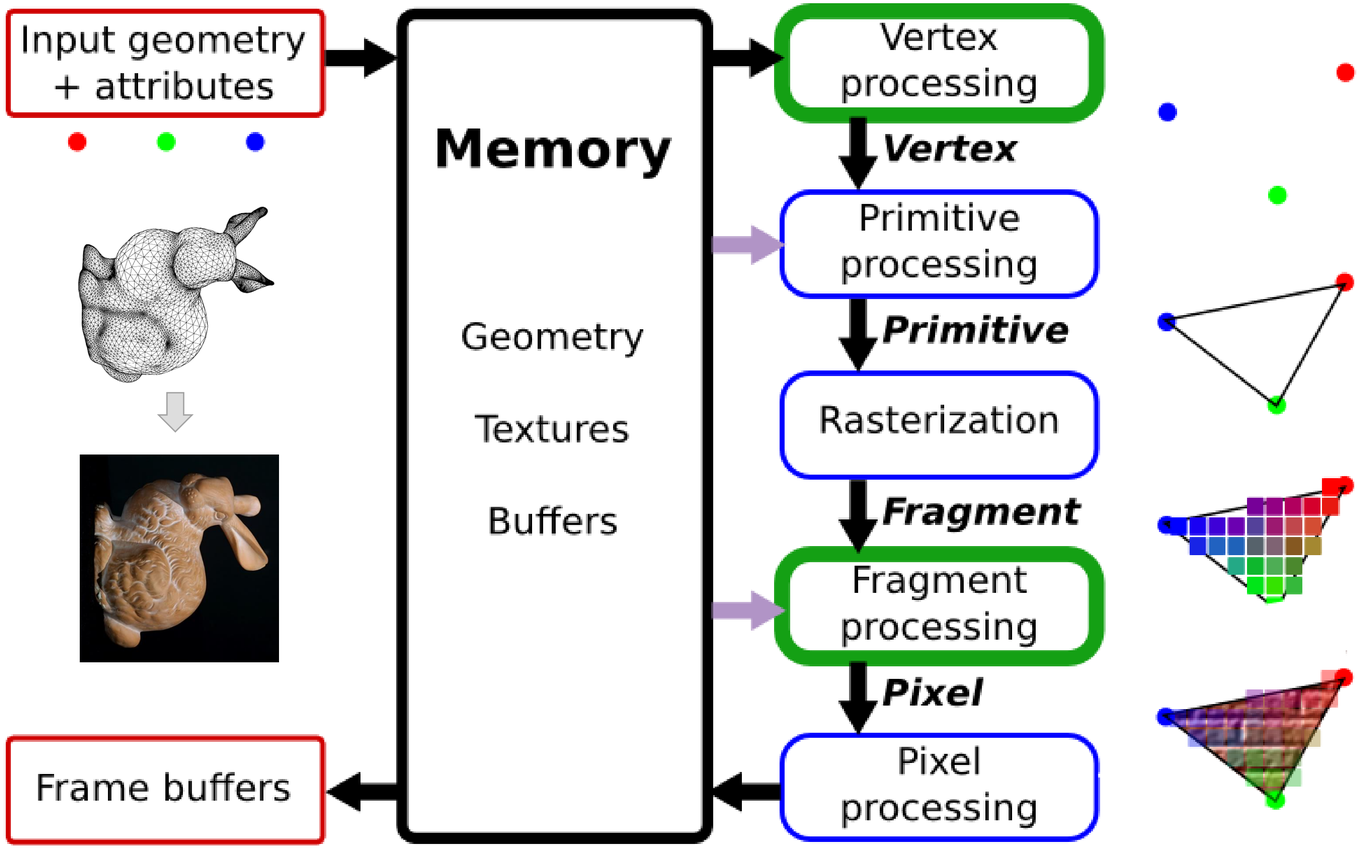


Fragment Processing on GPU

(VI) Convert these small pieces to screen pixels

Pixel Processing on GPU

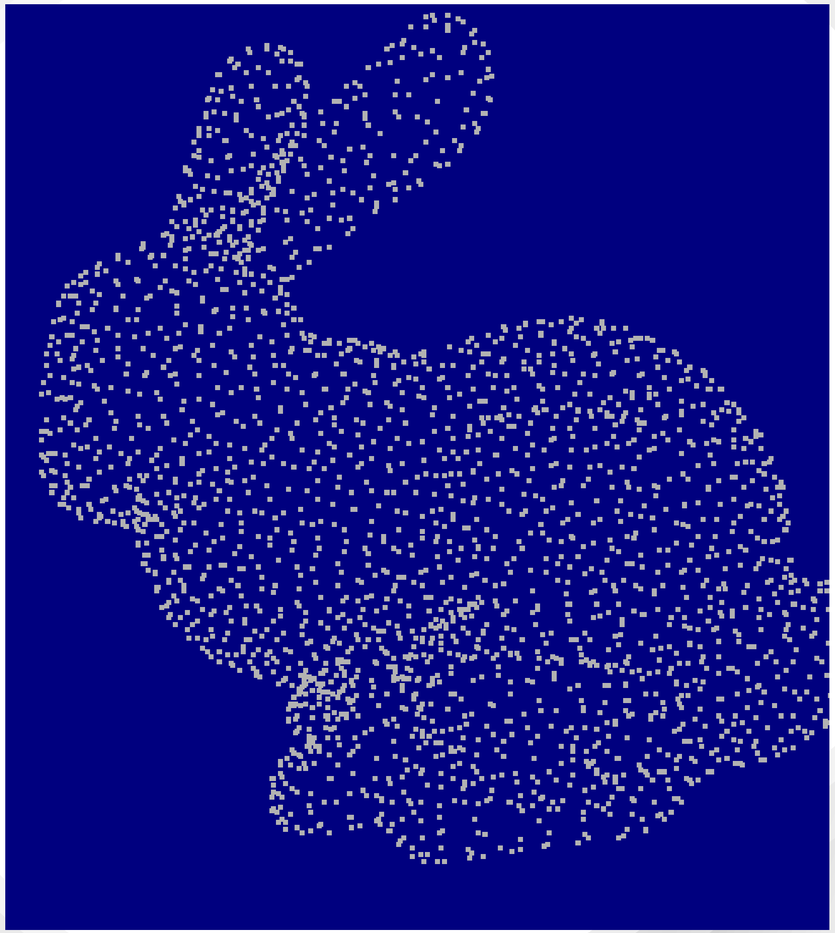
Quick Recap: The Full Pipeline



Now let's explain each stage with more technical details.

Stage 1: Vertex Processing

- The vertex processing stage processes vertex data through a vertex shader.
- Each vertex's properties, like **position**, **color**, and **texture coordinates**, are calculated and transformed to prepare for primitive processing.
- We can program this stage in **Vertex Shaders**.



We will study details in the lecture of transformation.

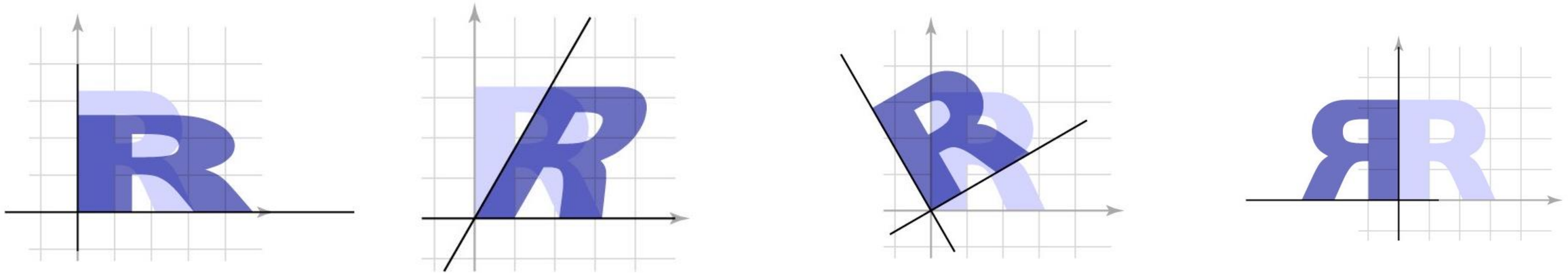
Vertex Operations

- Vertex operations typically involve matrix-vector multiplications.
- The two common operations are:
  - **Transforming an Object**: Applying model transformations such as translation, rotation, and scaling.
  - **Positioning for the Camera**: Adjusting the view to simulate a camera perspective.
- These operations are often combined into a single '**modelview**' matrix for efficiency.

$$\mathbf{p}_s = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{M}_{\text{cam}}\mathbf{M}_{\text{m}}\mathbf{p}_o$$

$$\begin{bmatrix} x_s \\ y_s \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{M}_{\text{m}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

An example of combined modelview matrix



Example of Vertex Operation: Transformation

- Create objects in convenient coordinates, and then transform into the scene later
- Make multiple copies of a single object
- Connect kinematics of linkages/skeletons for characters/robots
- We use matrices to transform vertices.
- For rendering numerous objects of the same shape but in different orientations, matrices are sent to the GPU to avoid redundant vertex storage.

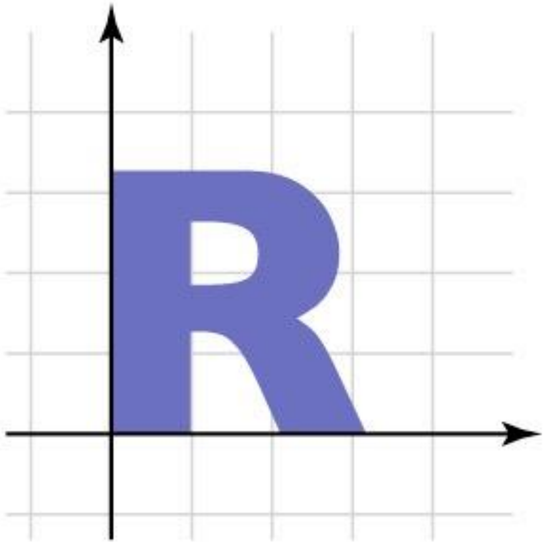
Example of Vertex Operation: Rotation

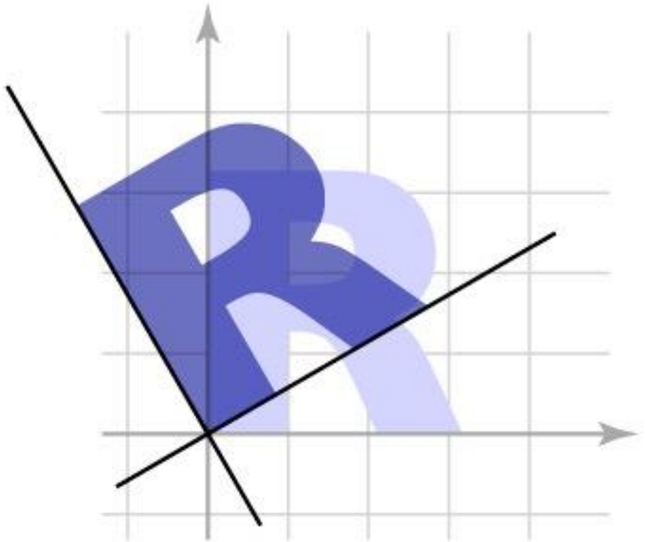
rotate **CCW** by angle θ

$$R_\theta \mathbf{p} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} p_x \cos\theta - p_y \sin\theta \\ p_x \sin\theta + p_y \cos\theta \end{bmatrix}$$

$$R_\theta^{-1} = R_{-\theta}$$

- A strategy: just memorize this matrix ¯_(ツ)_/¯
- Better strategy: understand why these are the columns



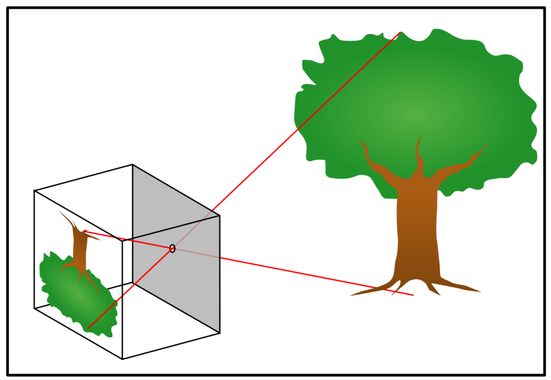


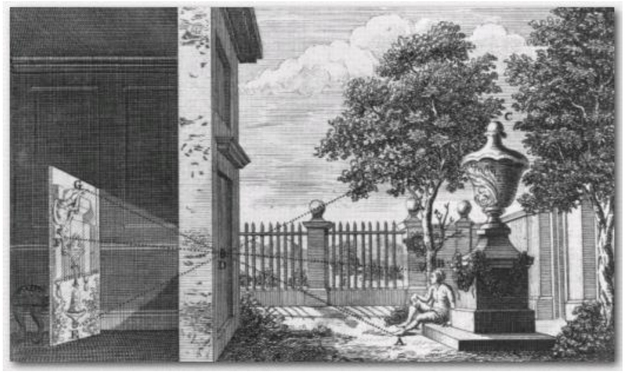
$$\begin{bmatrix} 0.866 & -0.05 \\ 0.5 & 0.866 \end{bmatrix}$$

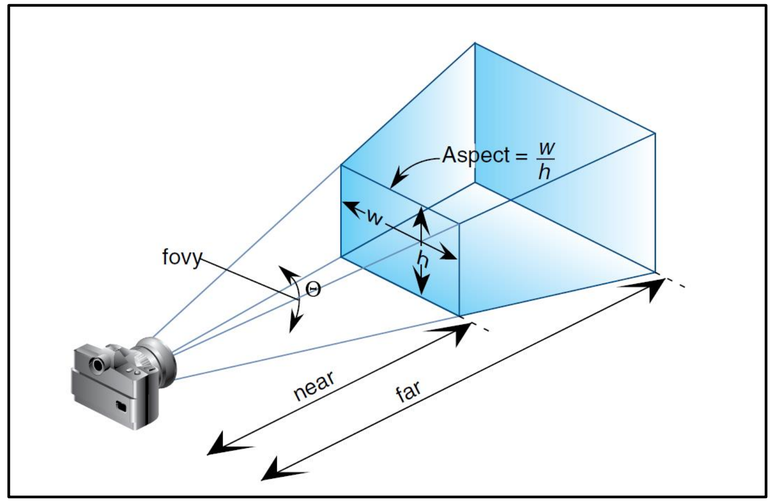
We will study details in the lecture of camera.

Camera

- The modern scanline renderer uses a pinhole camera.
- However, the image plane (i.e. the film) is placed in front of the pinhole, so that the image is not upside down
- The frustum is the volume of the view (shown in blue below)
- Notice the front and back clipping planes!
- The image plane (i.e. the film) is the clipping plane closest to the camera





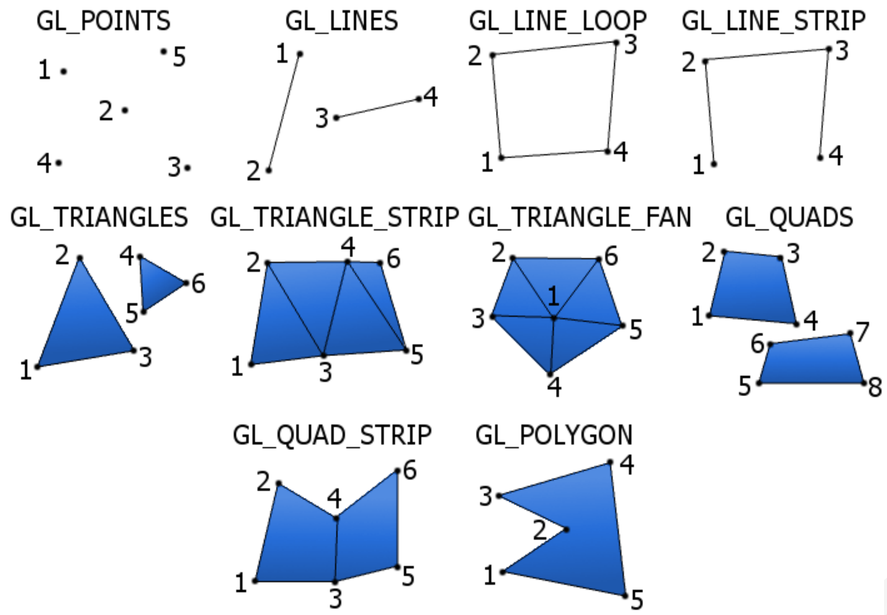


Pinhole camera

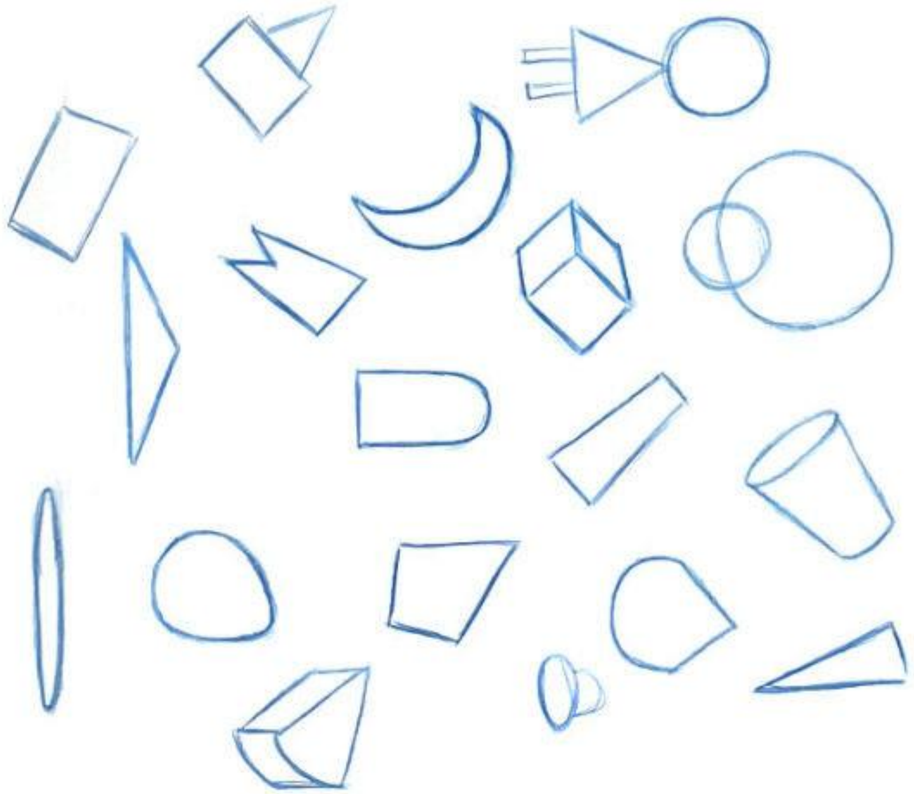
Camera model used in graphics pipeline

Stage II: Primitive Processing

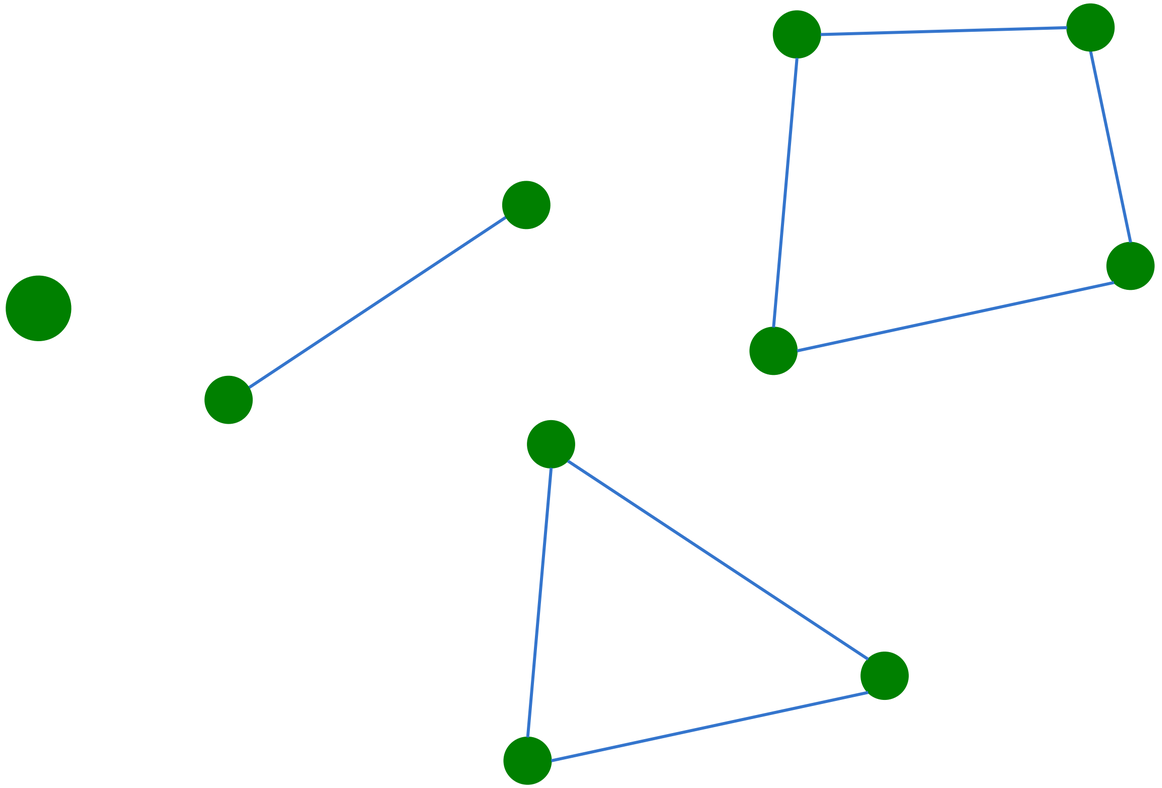
- Once vertices are processed, they are assembled into geometric primitives such as **points**, **lines**, or **triangles**.
- This stage may include tessellation, where primitives are subdivided into finer details, and geometry shading.
- This stage can be programmed using **Geometry Shaders**, which are advanced topics that we will not cover in this class.



Shape Primitives

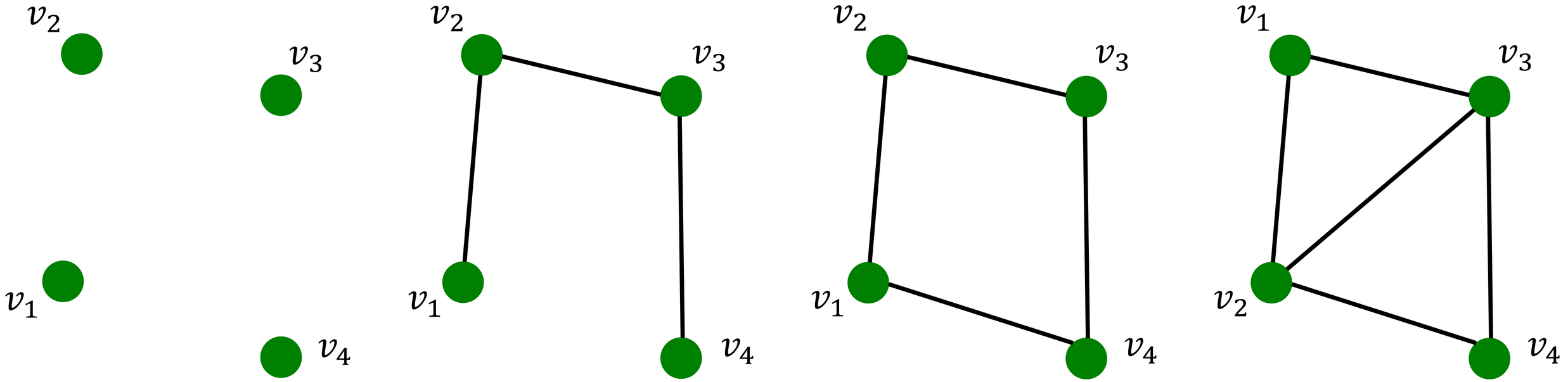


Primitives in Hand Sketching



Primitives in Computer Rendering

What shapes can be represented using four vertices?



GL_POINTS

GL_LINES

GL_QUAD

GL_TRIANLGE_STRIP

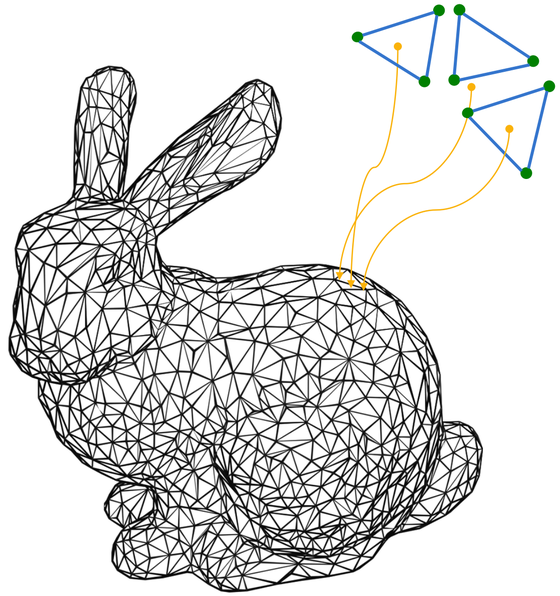
Input Vertex Array:

x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	x_4	y_4	z_4

We will study details in the lecture of mesh.

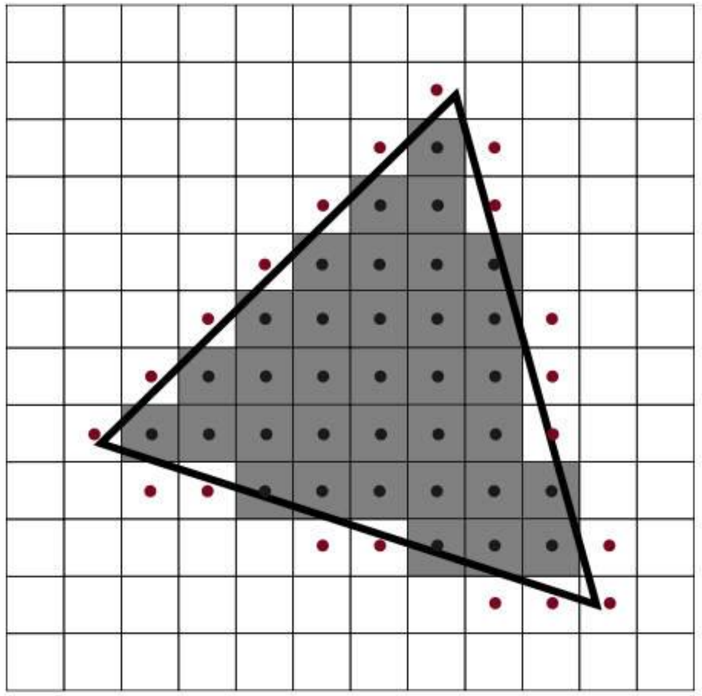
Triangle is the fundamental primitive for graphics!

- It's easy to break convex polygons into triangles
  - As a result, can focus on optimizing the hardware/software implementation of only one primitive (i.e. optimize for triangles)
- Triangles have some nice properties
  - Guaranteed to be planar (unlike quadrilaterals)
  - Guaranteed to have a well-defined interior
  - There exists a well-defined method for interpolating values in the triangle interior (barycentric interpolation)
  - We will study many triangle-related algorithms in this class!



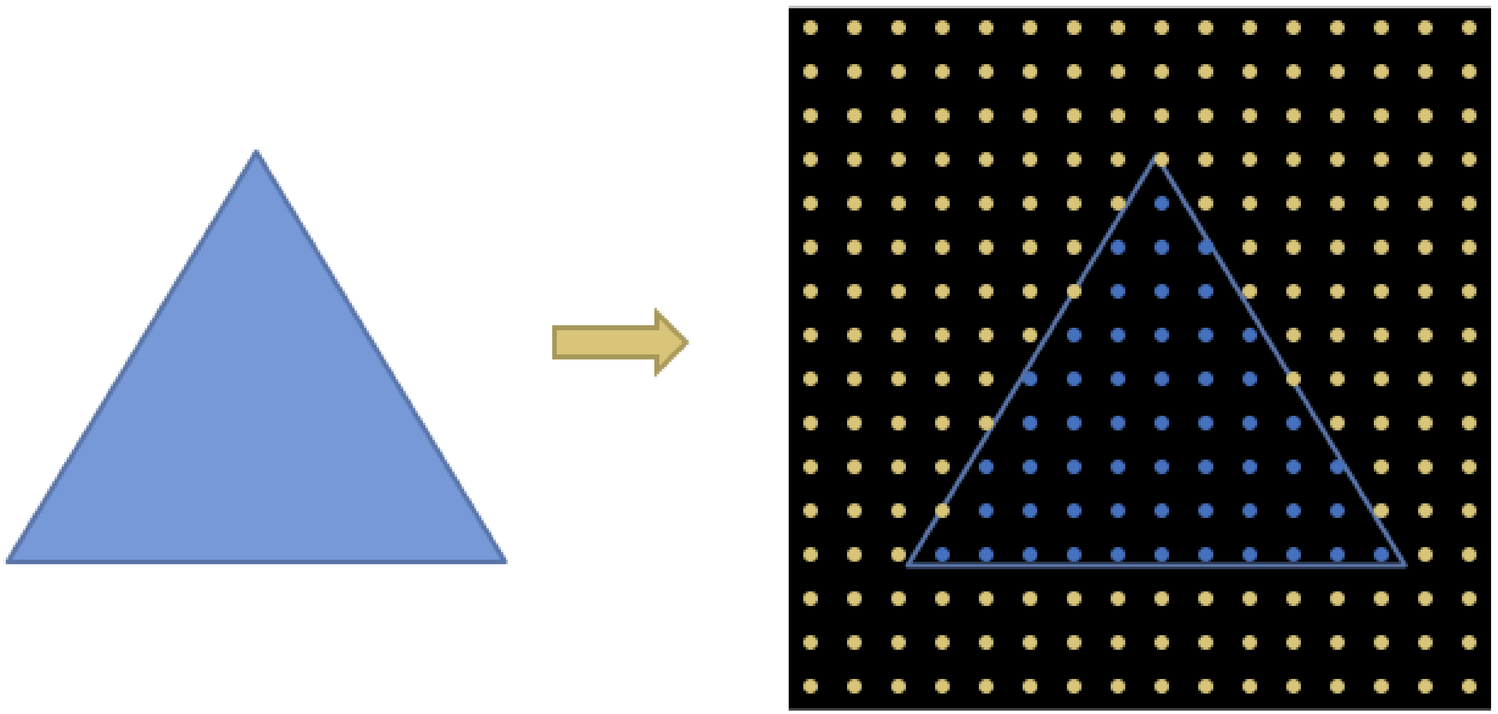
Stage III: Rasterization

- Rasterization converts primitives into a 2D image or framebuffer.
- It determines the pixels that form the primitives, performing interpolations for pixel positions. This stage sets the scene for fragment processing.
- This stage is **not programmable**.



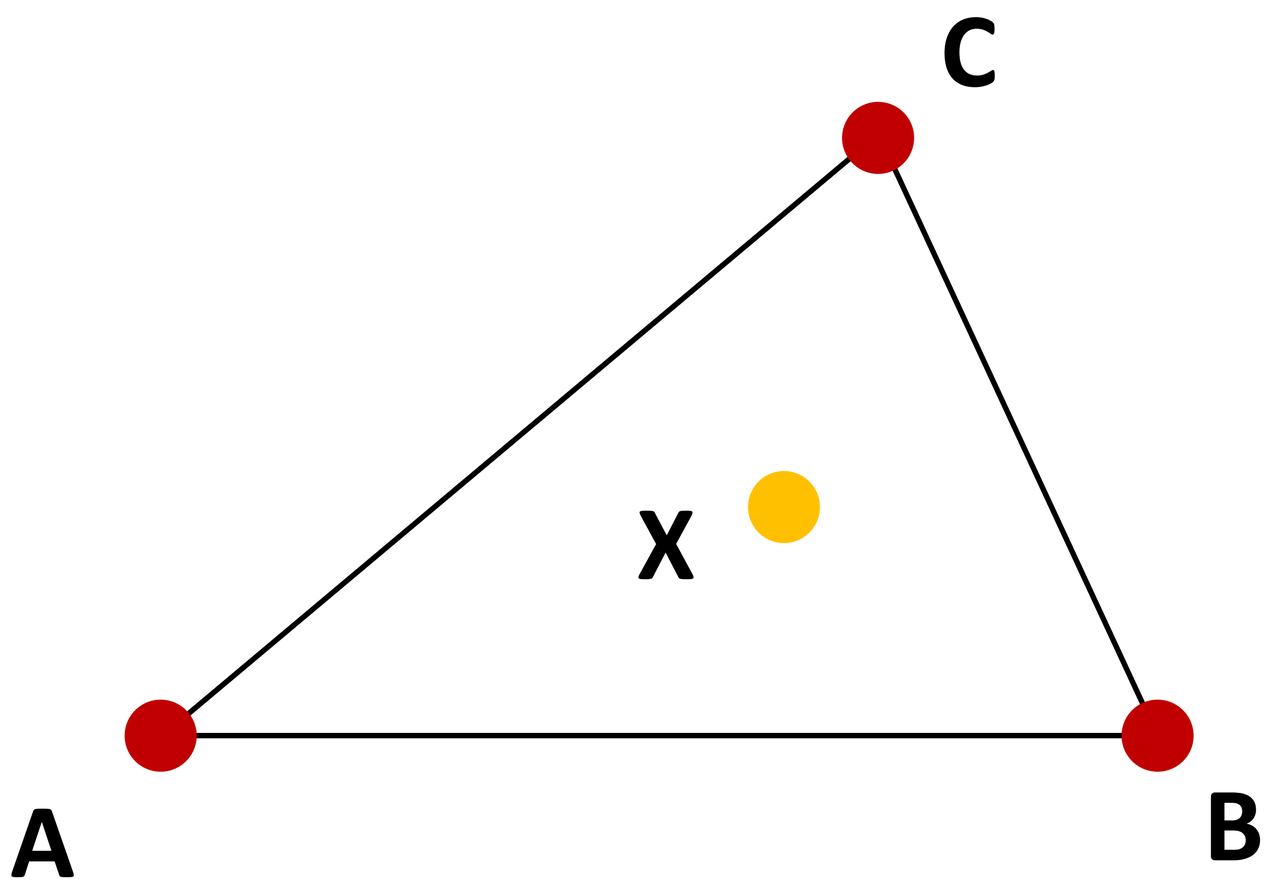
Rasterization

- Rasterize 2D triangles after transforming the vertices to screen space
- Color the pixels inside the triangle with the RGB-color of the triangle



Question: How do we determine if a point is within a triangle or not on the screen?

Check point inside/outside using cross product



$$(B - A) \times (X - A) > 0 \;\&\&\; (C - B) \times (X - B) > 0 \;\&\&\; (A - C) \times (X - C) > 0$$

2D Cross Product

- Definition:

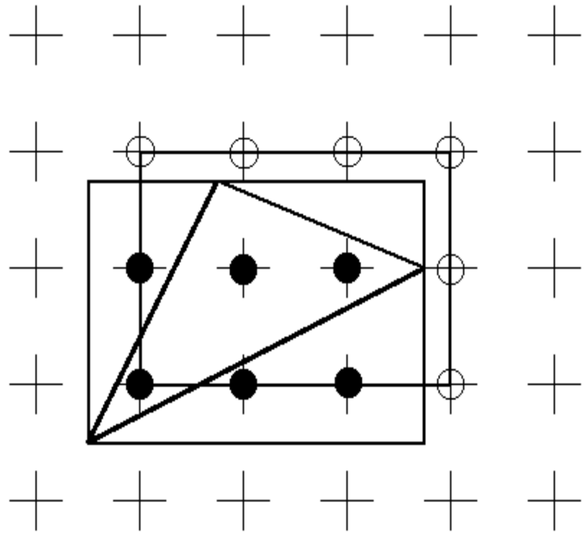
$$\vec{a} \times \vec{b} = \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix} = a_x b_y - a_y b_x$$

- The sign of the scalar (positive or negative) indicates the orientation of the two vectors:
  - A positive value means b is counterclockwise from a
  - A negative value means b is clockwise from a

Bounding Box Acceleration

- Inefficient to check every pixel on the screen
- Calculate a bounding box around the triangle, and only check pixels inside the box
- Round coordinates upward (ceil) to the nearest integer

```
bound3( vert v[3], bbox& b )
{
  b.xmin = ceil(min(v[0].x, v[1].x, v[2].x));
  b.xmax = ceil(max(v[0].x, v[1].x, v[2].x));
  b.ymin = ceil(min(v[0].y, v[1].y, v[2].y));
  b.ymax = ceil(max(v[0].y, v[1].y, v[2].y));
}
```

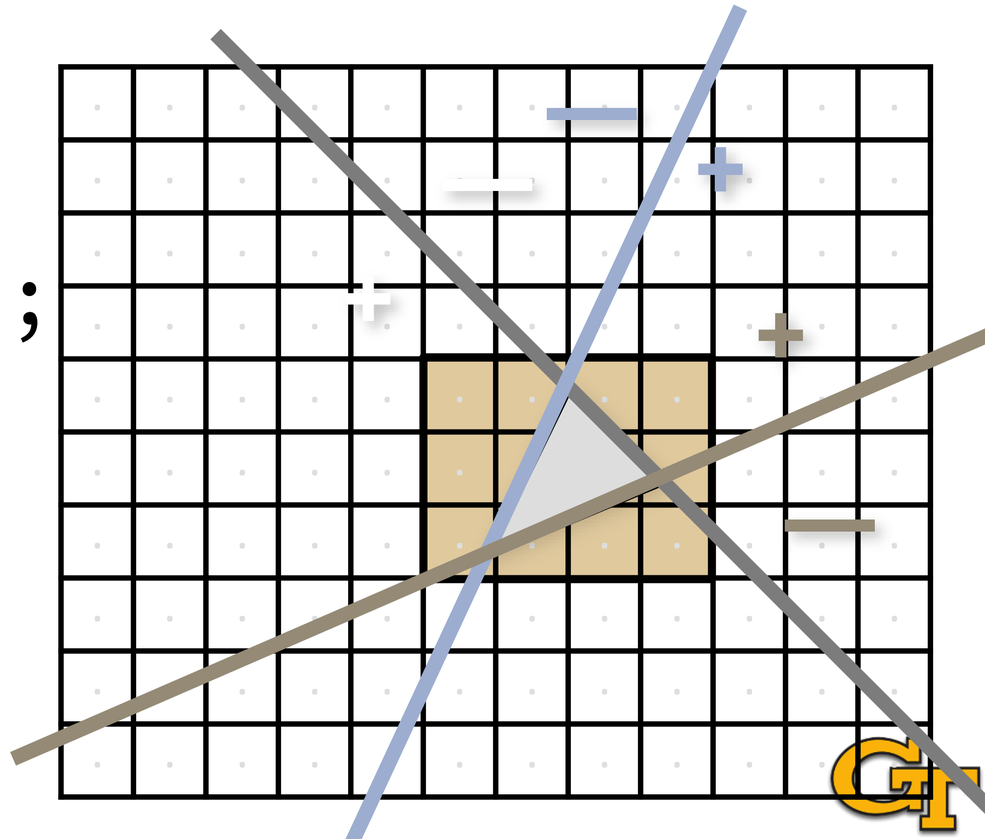


Test points with filled circles
Don't test hollow circles

Rasterization Pseudocode

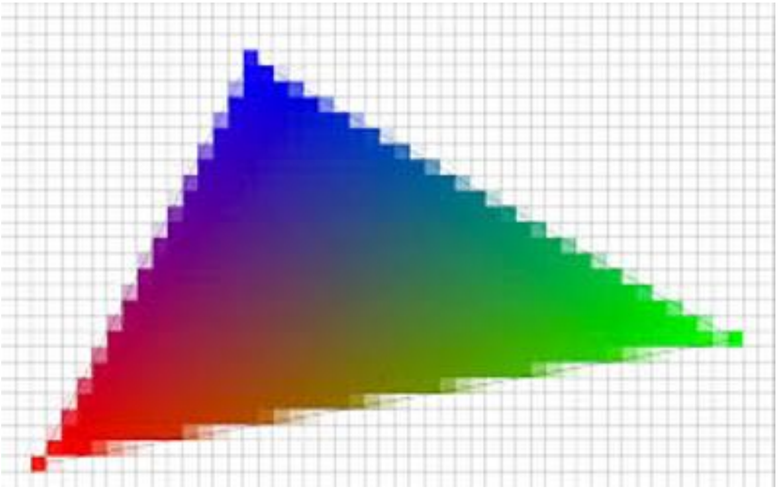
```
for every triangle
    project vertices, compute the Eᵢ
    compute bbox, clip box to screen limits
    for all pixels in box
        evaluate cross products
        if all > 0
            framebuffer[x,y] = c;
```

Bounding box clipping is easy, just clamp the coordinates to the screen rectangle



Stage IV: Fragment Processing

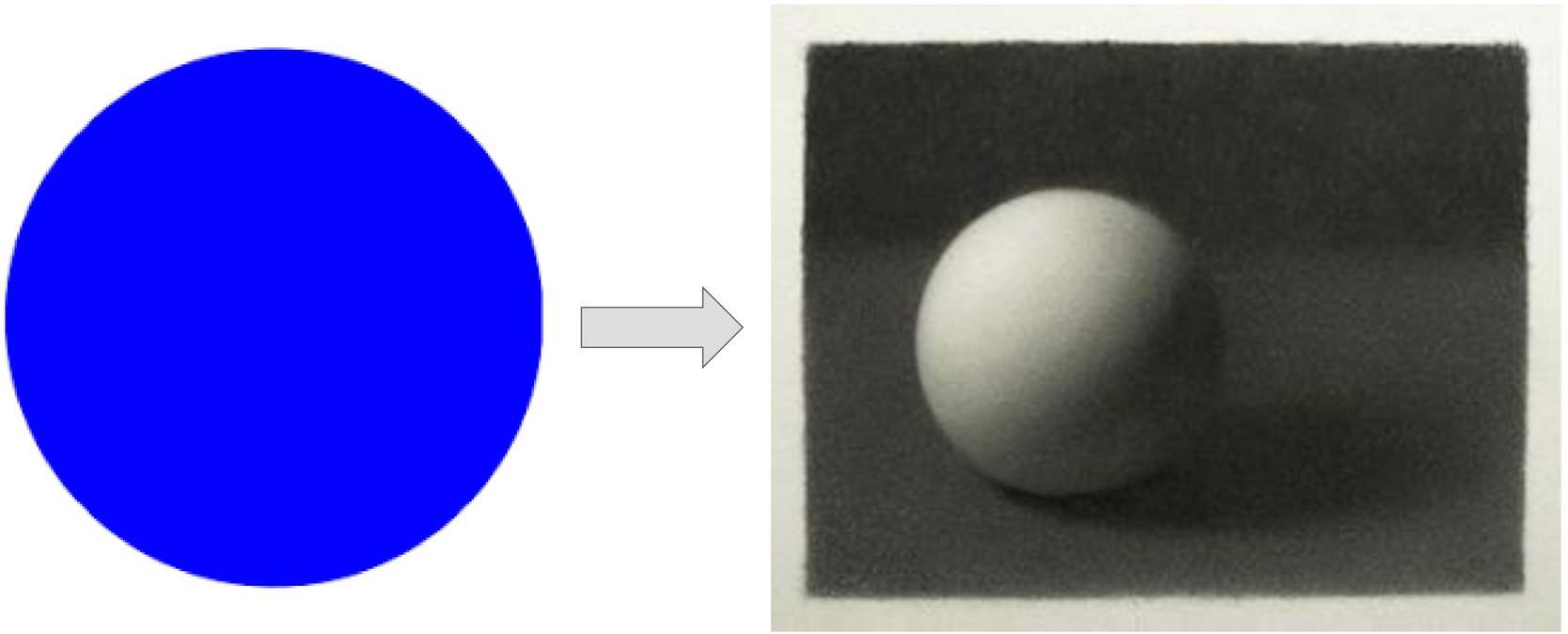
- Fragments can be thought of as pixels **(but they are actually not!)**, are processed to determine their final color and properties.
- Fragment shaders compute lighting, apply textures, and carry out operations for visual effects.
- We can program this stage in **Fragment Shaders**.



We will study details in the lecture of lighting & shading.

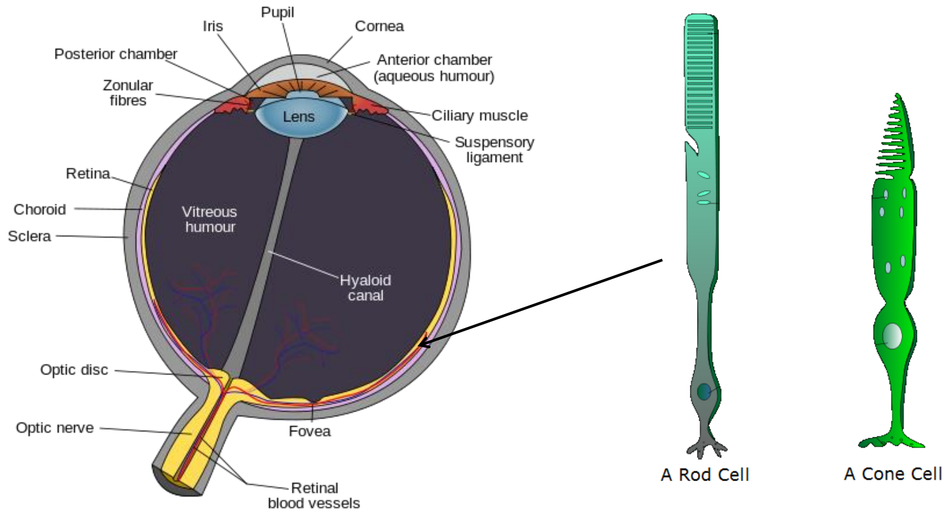
Lighting & Shading

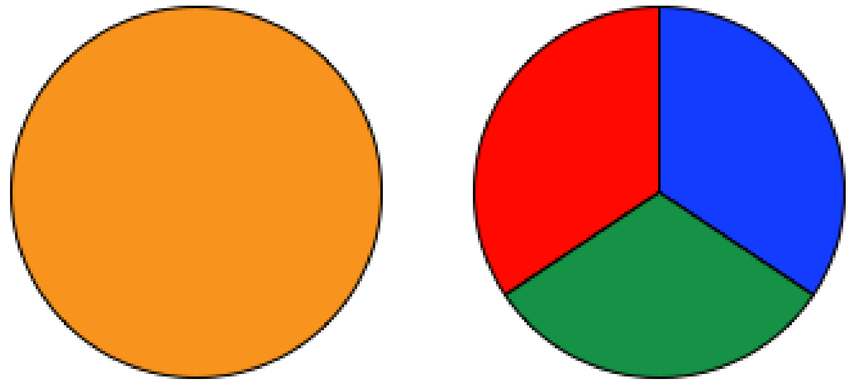
- We calculate the fragment color according to their position, shape, and light sources
  - Without lighting & shading, you'd get a 2D splatted cartoon view of your objects
  - E.g., fragment shader turns a 2D circle into a 3D sphere



Human Eye and Color Matching Experiment

- Human retina has two kinds of sensors
  - Rods only measure light intensity
  - 3 different cones measure three colors (red, green, blue)
- Adjust the brightness of three single wavelength lasers until a human observer mistakenly thinks it matches another color C
  - Result: humans mistakenly believe that all "colors" are matched by certain combinations of three single wavelength lasers
  - This makes sense because each of the three cones can only send a single signal





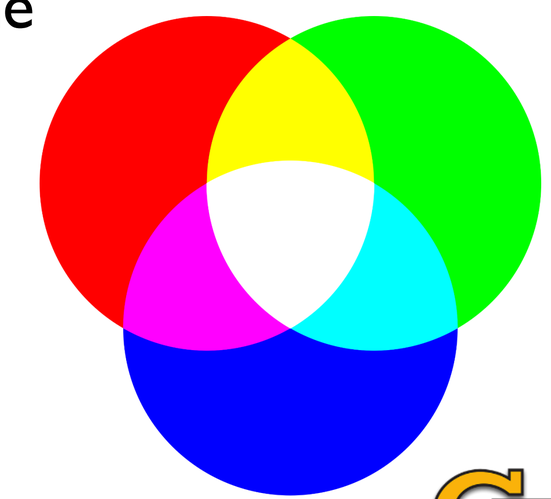
C "=" R "+" G "+" B

RGB Color

- Since the human eye works with only 3 signals (ignoring rods), we work with 3 signals for images, printers, and displays.

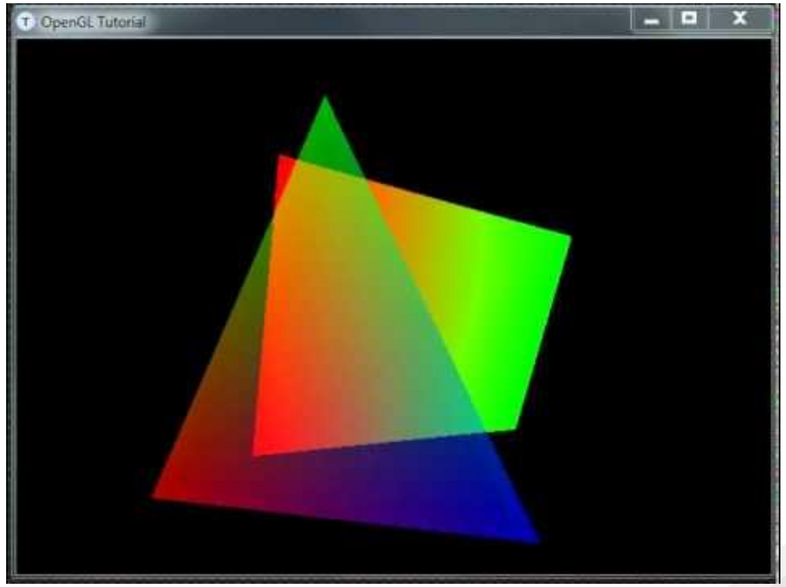
- Image formats store values in the R, G, and B channels
  - The values stored are typically between 0 and 1 in OpenGL
  - In other areas such as image processing or computer vision, the range is between [0, 255]

- For instance, (1,0,0) is red, (0,0,1) is blue (1,1,0) is yellow, (1,1,1) is white, etc.



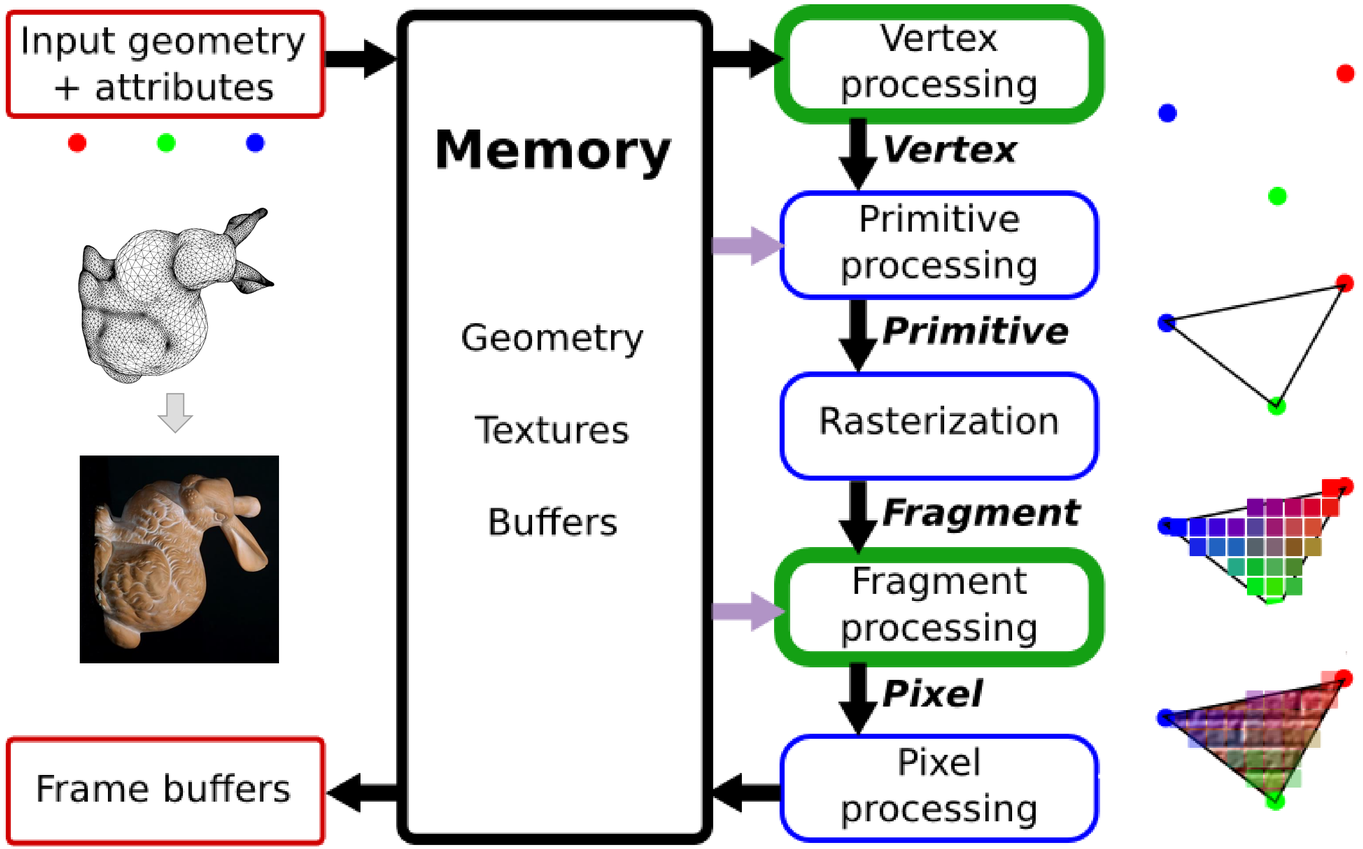
Stage V: Pixel Processing

- The pixel processing stage involves operations like **blending** and testing pixels for **depth** and **stencil** comparison.
- The final pixel colors are output to the screen, resulting in the rendered image.
- This stage is not programmable.



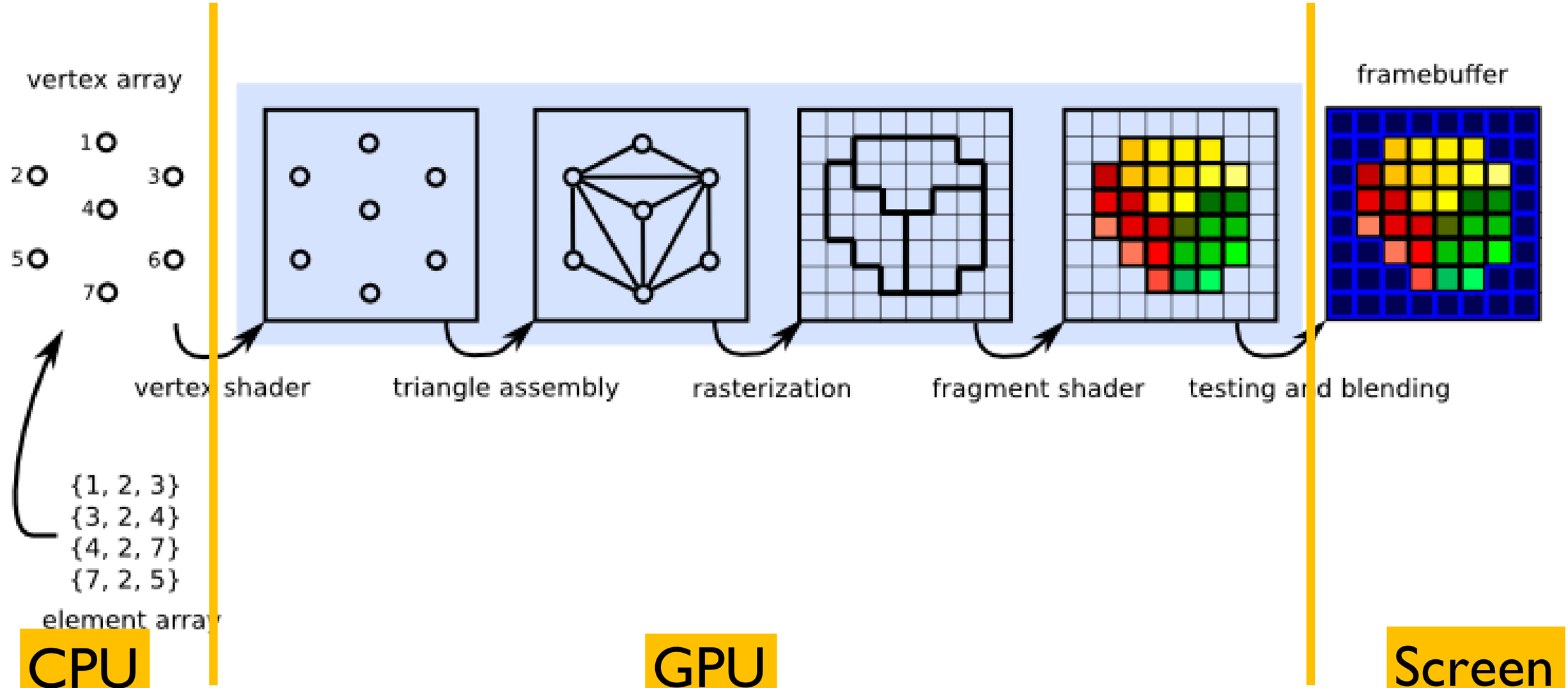
Key Takeaways

- **Vertex properties (e.g., position, color) and primitive properties (e.g., triangle indices) are pipeline inputs**
- **Pixel colors are pipeline output**
- **Only the vertex processing and fragment processing are programmable**

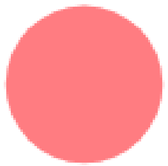


CPU-GPU Data Communication

Which stages are performed by the CPU versus the GPU?



Goal: Drawing a Bouncing Sphere on Screen



Example: Drawing a Bouncing Sphere

CPU

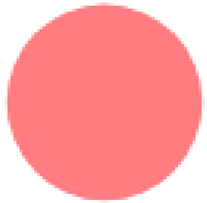
- v+=f/m*dt
- x+=v*dt
- for each vertex x_i:
- x_i=x_i0+x

GPU

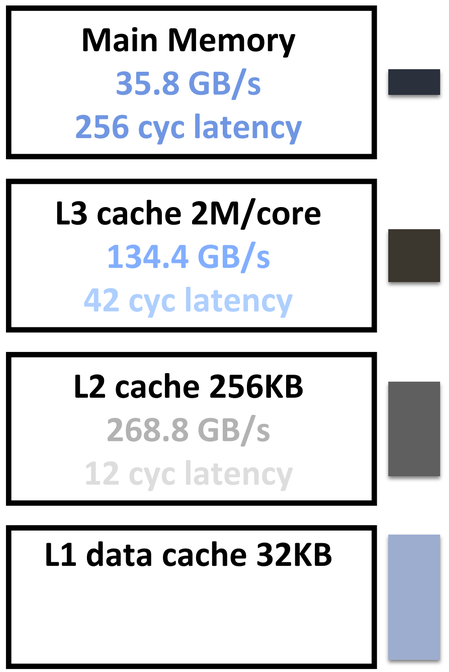
- Send x_i to GPU
- Go through the five GPU stages

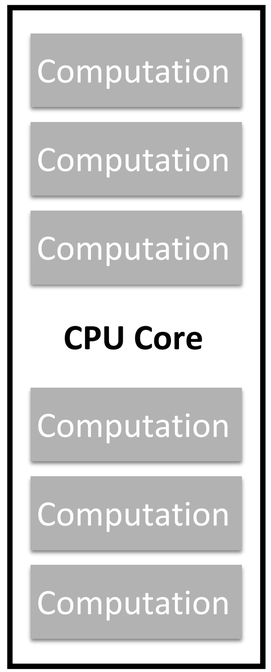
Screen

- Draw x_i on Screen



How fast are the data transfers?



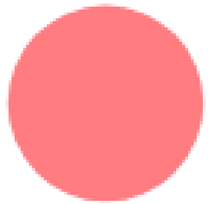


* **The state-of-the-art NVLink CPU-GPU data rate: 35.3 GB/s**
* **PCI CPU-GPU data rate: 5 GB/s**

Data transfer from CPU to GPU is *SUPER* slow!

A Better Way to Improve Data Transfer Performance

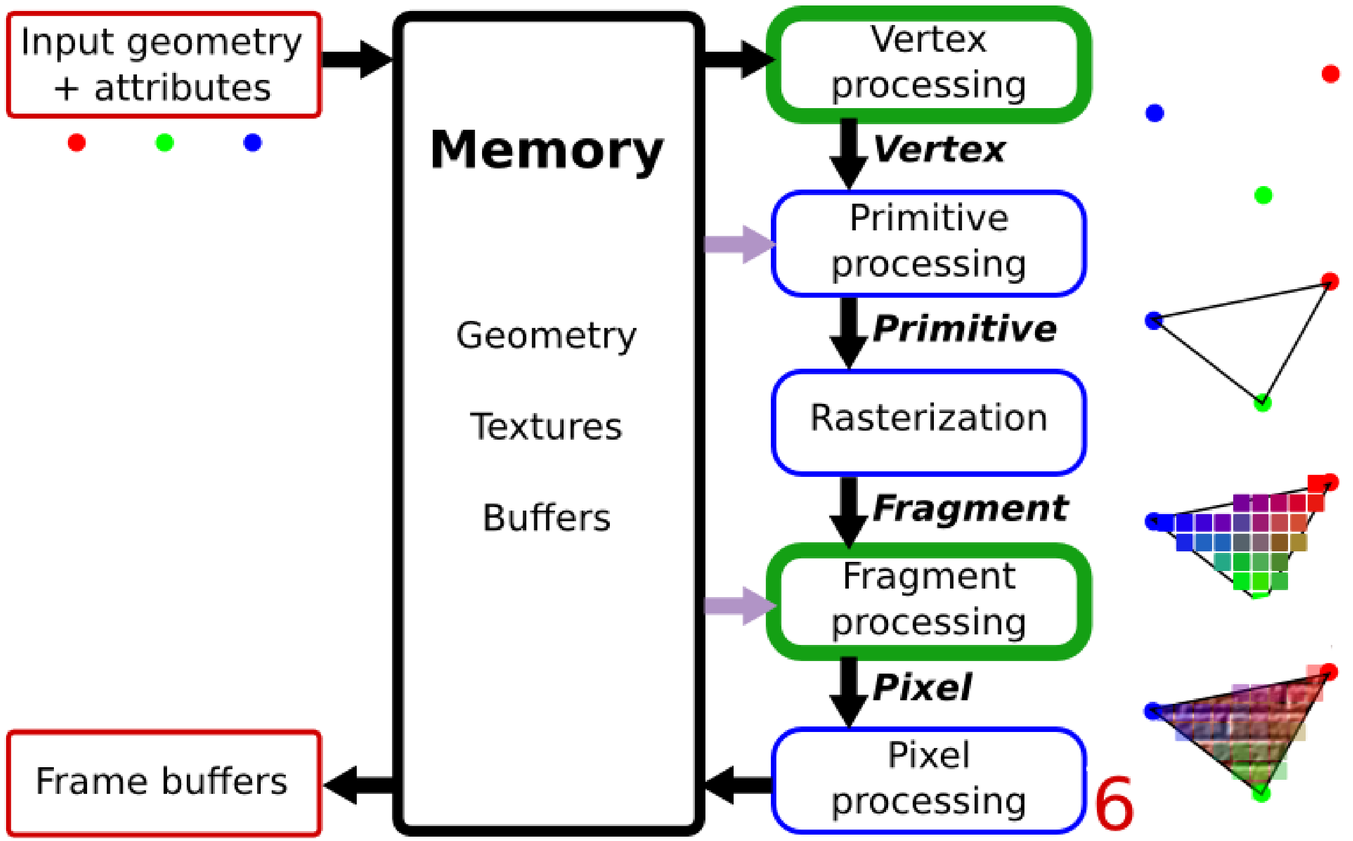
CPU	• v+=f/m*dt • x+=v*dt • for each vertex x_i: • x_i=x_i0+x • send x_i to GPU • draw x_i	○ v+=f/m*dt ○ x+=v*dt ○ build a transformation matrix M based on x ○ **send M to GPU**
GPU		○ for each vertex x_i: ○ x_i=M*x_i0
Screen		○ draw x_i



CPU vs GPU Workloads

- **Latency-sensitive (CPU tasks, e.g., physics simulation)**
  - Long programs with serial dependencies
  - Complex data-dependent control and memory access patterns
  - Few independent work items
    - *Not* 2 million pixels
- **Throughput-sensitive (GPU tasks, e.g., graphics rendering)**
  - Large number of independent but similar work items
  - Heavy on arithmetic (lots of math/memory op)
  - Coherent control, little data-dependent branching
  - Coherent memory accesses

Summery: Render a Triangle on Screen



- **Send triangle vertices from CPU to GPU**
- **Transform and project triangle vertices to 2D screen**
- **Rasterize triangle: find which fragments should be lit**
- **Compute fragment color**
- **Test visibility (Z-buffer), update pixel colors in frame buffer color**