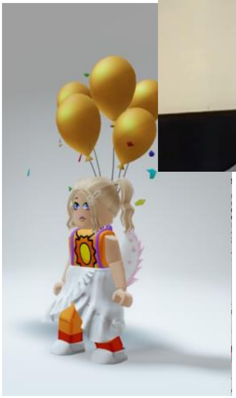
TDA362/DIT224 – Computer Graphics



+

=>

Starting 10:00 …

Teacher: Ulf Assarsson

Chalmers University of Technology

This Course

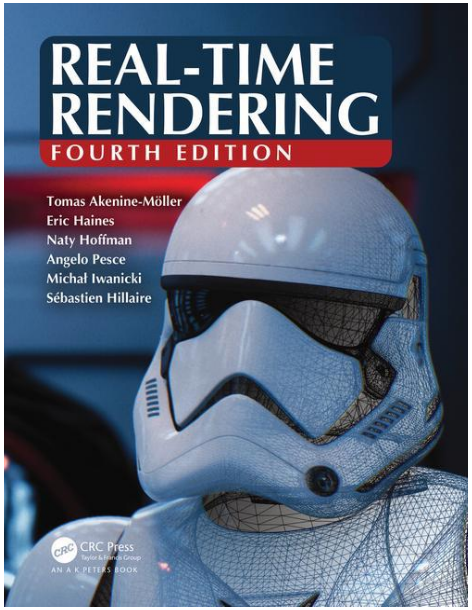
- Algorithms!

Real-time Rendering


Understanding Ray Tracing

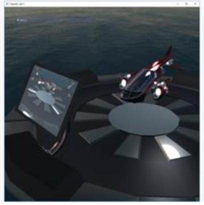
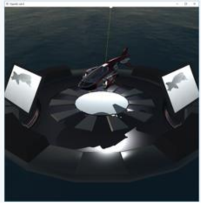
Course Info

- Study Period 2 (lp2)
- Real Time Rendering, 4th edition
  - Available on Cremona at discount.
- Schedule:
  - Mon 13-15, w2 only
  - Tues 10-12,
  - Fri 9-12,
    - ~14 lectures in total, ~2 / week
  - Lab slots:
    - Mon: 17-21
    - Tues: 13-21
    - Wed: 13-21
    - Thur: 9-12 + 17-21
- Homepage:
  - Google "TDA362" or
  - "Computer Graphics Chalmers"







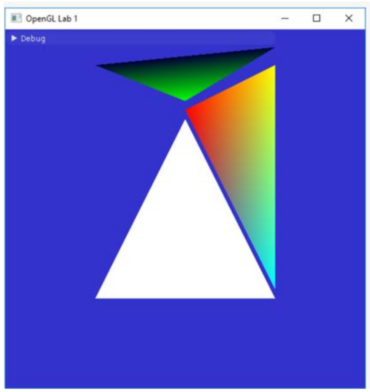
Tutorials

- All laborations are in C++ and OpenGL
  - Industry standard
  - No previous (C++) knowledge required
- Six shorter tutorials that go through basic concepts
  - Basics, Textures, Camera&Animation, Shading, Render-to-texture, Shadow Mapping
- One slightly longer lab where you put everything together
  - Real-time rendering

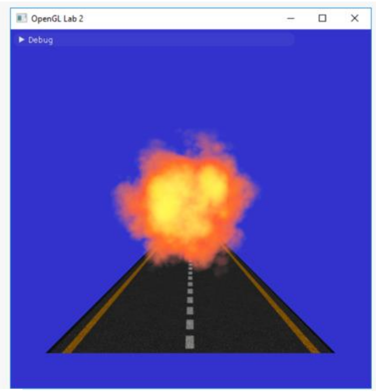
    or
  - Path tracer

Tutorials 1-6

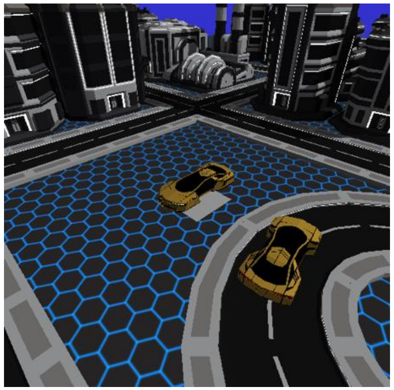
Don't just copy/pase ☺



Rendering a triangle

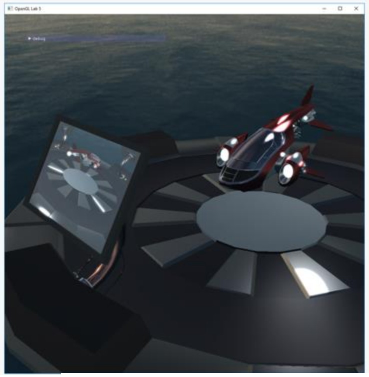


Textures

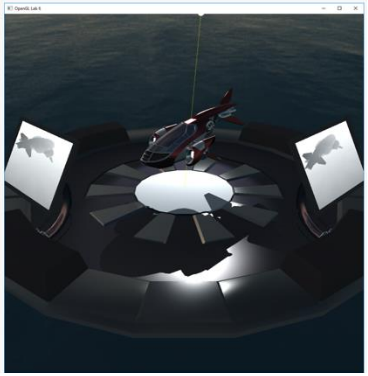


Animation

Shading



Render to textures



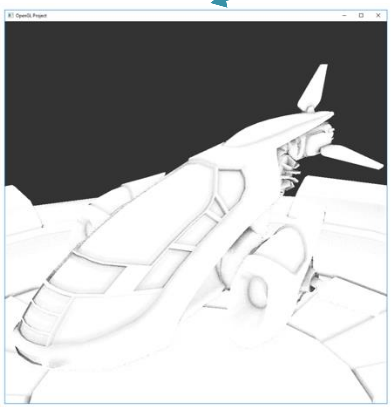
Shadow maps

Project

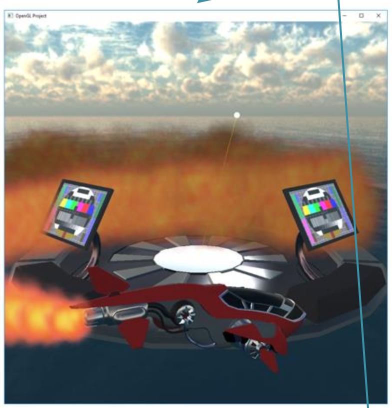
Choose at least 1 from:

Project

Real-time rendering or Offline rendering

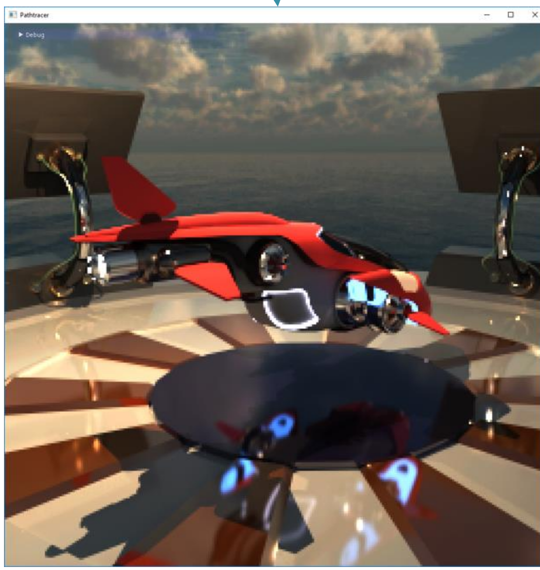


Screen-space ambient occlusion



Particle System

Height field



Path Tracing

Custom environment map (difficult)

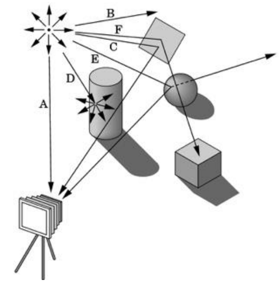
Tutorials

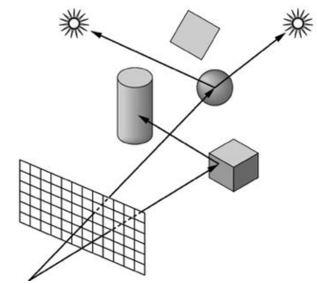
- Info: http://www.cse.chalmers.se/edu/course/TDA362/tutorials.html
- To pass the tutorials:
  - Present your solutions to lab assistant.
  - Deadlines:
    - Lab 1+2+3: Thursday week 2.
    - Lab 4 : Thursday week 3.
    - Lab 5+6: Thursday week 4
    - Lab 7 / Project: Thursday week 7.
- Do the tutorials in groups (Labgrupper) of two, or individually if you prefer.
- First deadline: Thurs. next week.

Computer Graphics:
– two main principles…

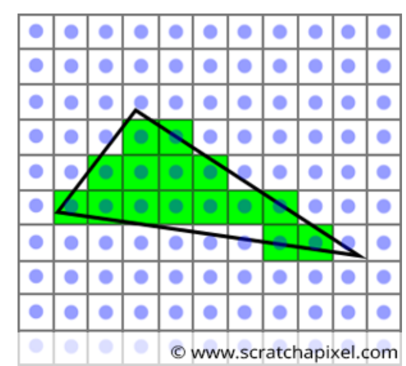
…for computer-generating the appearance of a virtual 3D scene:

- Ray Tracing:
  - **Forward** ray tracing: Tracing light beams from light sources and how they reach the virtual camera.
  - **Backward** ray tracing: Tracing the light beams backwards, i.e., from the camera and all the way back to the light sources.





- Rasterization:
  - Draw the scene triangles one by one onto the pixels of the screen and, for each pixel, compute the color (by regarding light sources and perhaps also surrounding objects).



Forward Ray Tracing

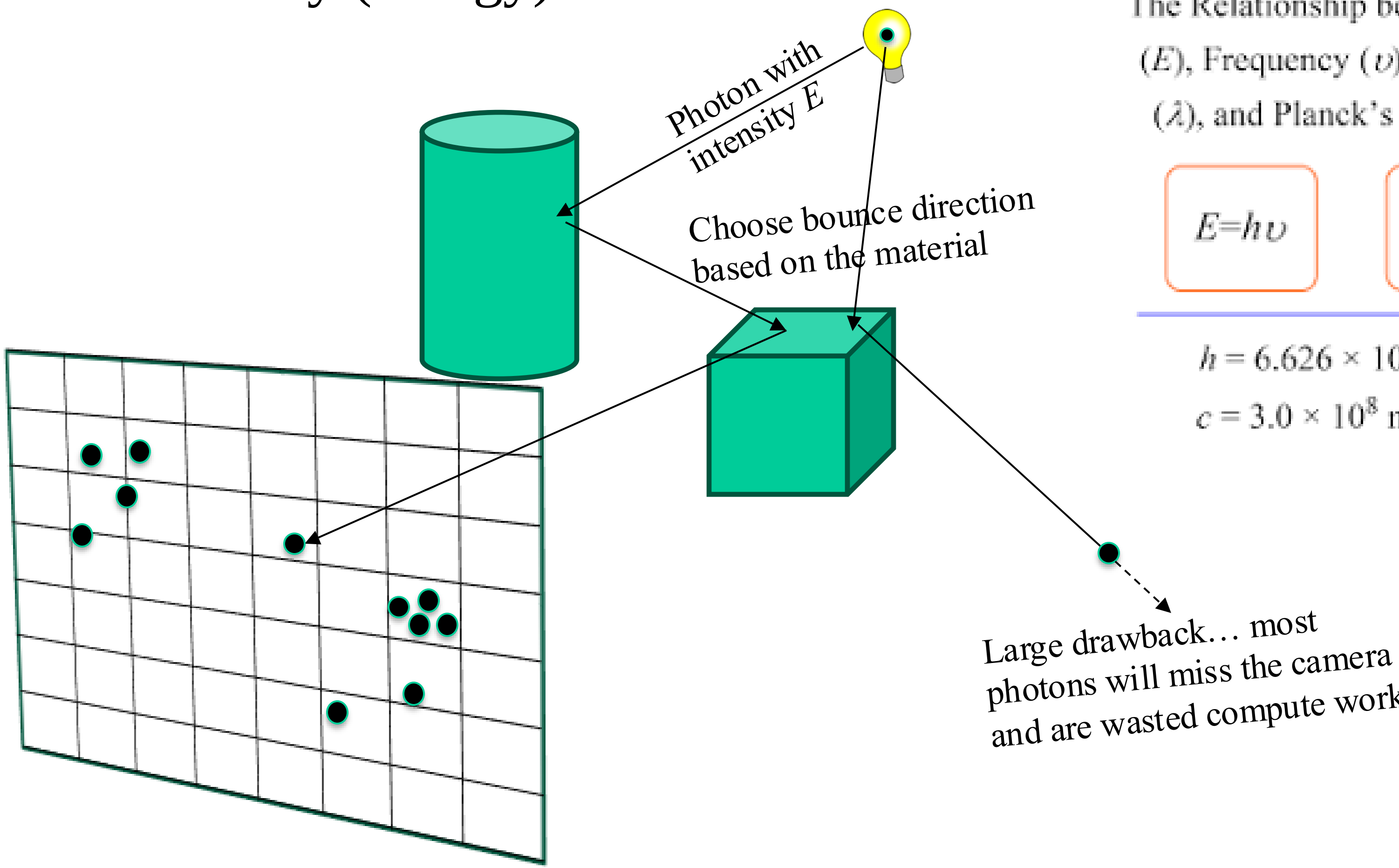
Forward ray tracing is simple and automatically gives correct intensity (energy) distribution on screen.

The Relationship between Energy (E), Frequency (υ), Wavelength (λ), and Planck's Constant (h)

$$E = h\upsilon \qquad E = \frac{hc}{\lambda}$$

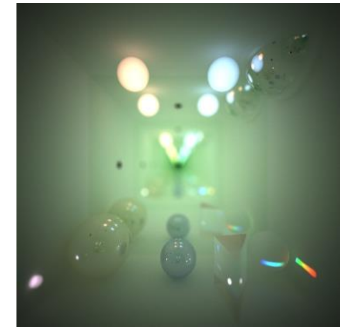
$h = 6.626 \times 10^{-34}$ J·s

$c = 3.0 \times 10^{8}$ m/s

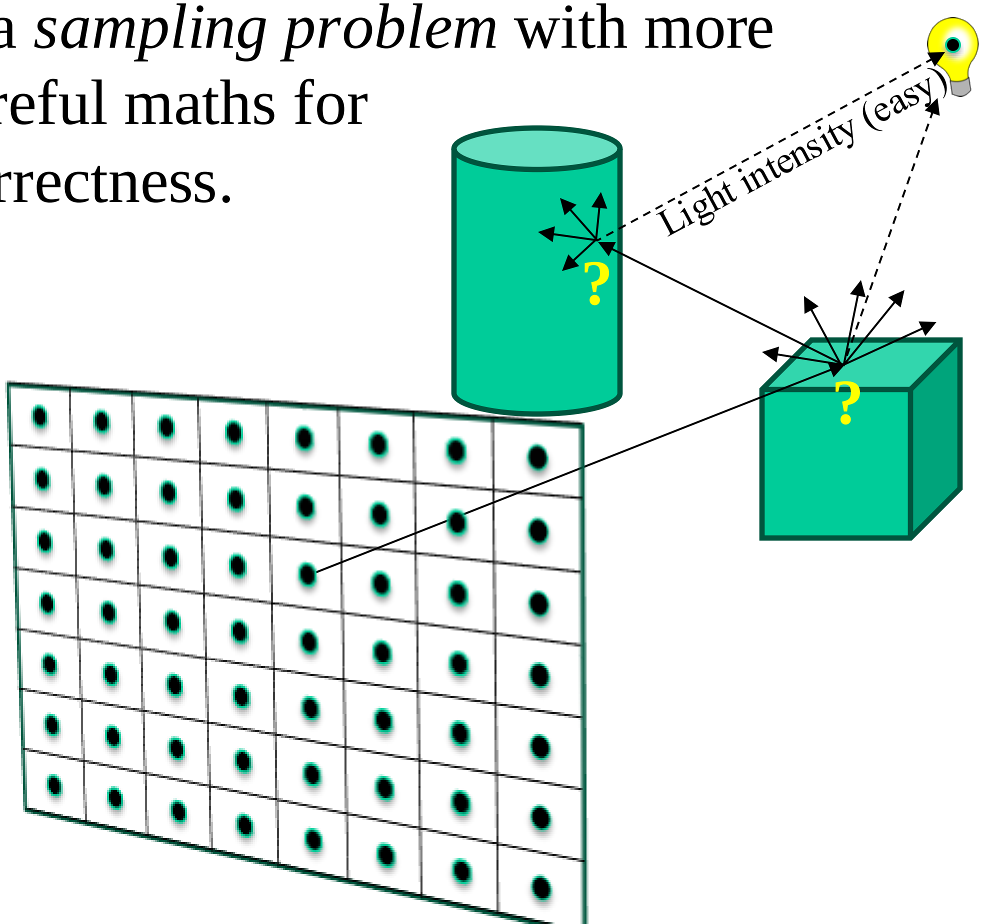


"Trace some trillion photons and you probably have a good image."

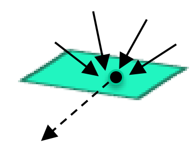
Backward Ray Tracing



Backward ray tracing is more efficient, but finding correct intensity (energy) and relevant incoming light directions is a *sampling problem* with more careful maths for correctness.

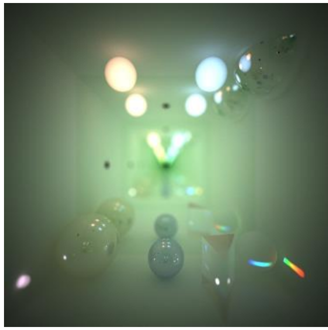


? = At each backward-ray hjtpoint, , based on the material, sample incoming light, by shooting rays, to estimate outgoing light to pixel **intensity** (or energy) to pixel

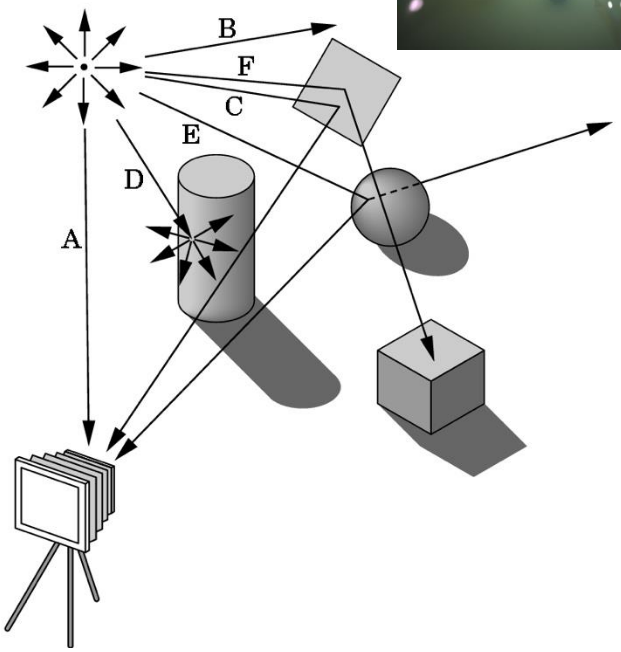


"Trace a billion rays backwards and you probably have a great image."

Forward Ray Tracing



One way to form an image is to follow rays of light (or photons) from a point source finding which rays enter the lens of the camera. Each ray of light may have multiple interactions with objects before being absorbed or going to infinity.



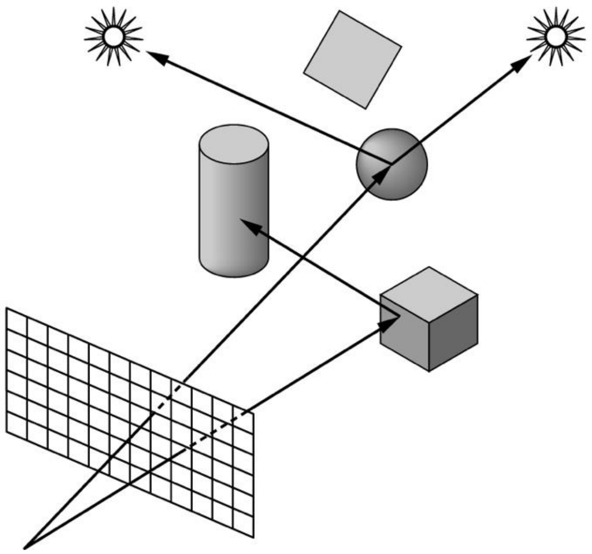
Pros: Algorithmically very easy to generate physically correct images.

Cons: Extremely slow. Only few of the traced rays will reach the camera sensor and actually contribute to the image.

"Trace some trillion photons and you probably have a good image."

Backward Ray Tracing

- Follow rays of light backwards, i.e., from the camera sensor (center of projection) into the scene until they either are absorbed by objects or go off to infinity.
  - At each bounce position, estimate incoming light intensity and color by following possible bounce directions, given the material.
  - Cons: Complicated but possible to get accurate convergence. We use Monte-Carlo sampling theory from maths for how to best sample an unknown signal. E.g., we do not know photon density nor from which direction the photon came. Combinations of forward + backwards ray tracing can be used to remedy this.
  - Pros: Faster but still slow compared to rasterization

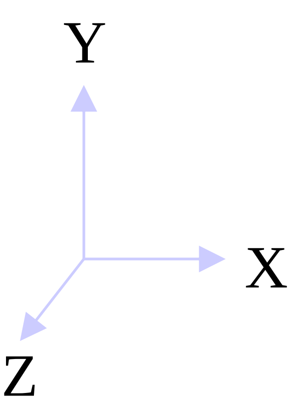


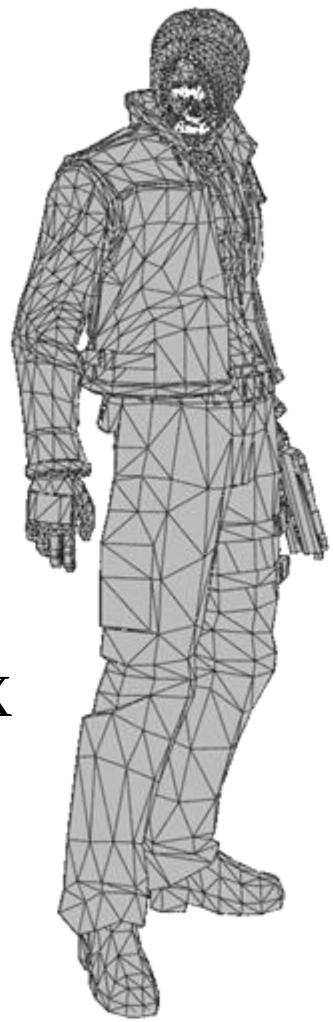
“Trace a billion rays backwards and you probably have a great image.”

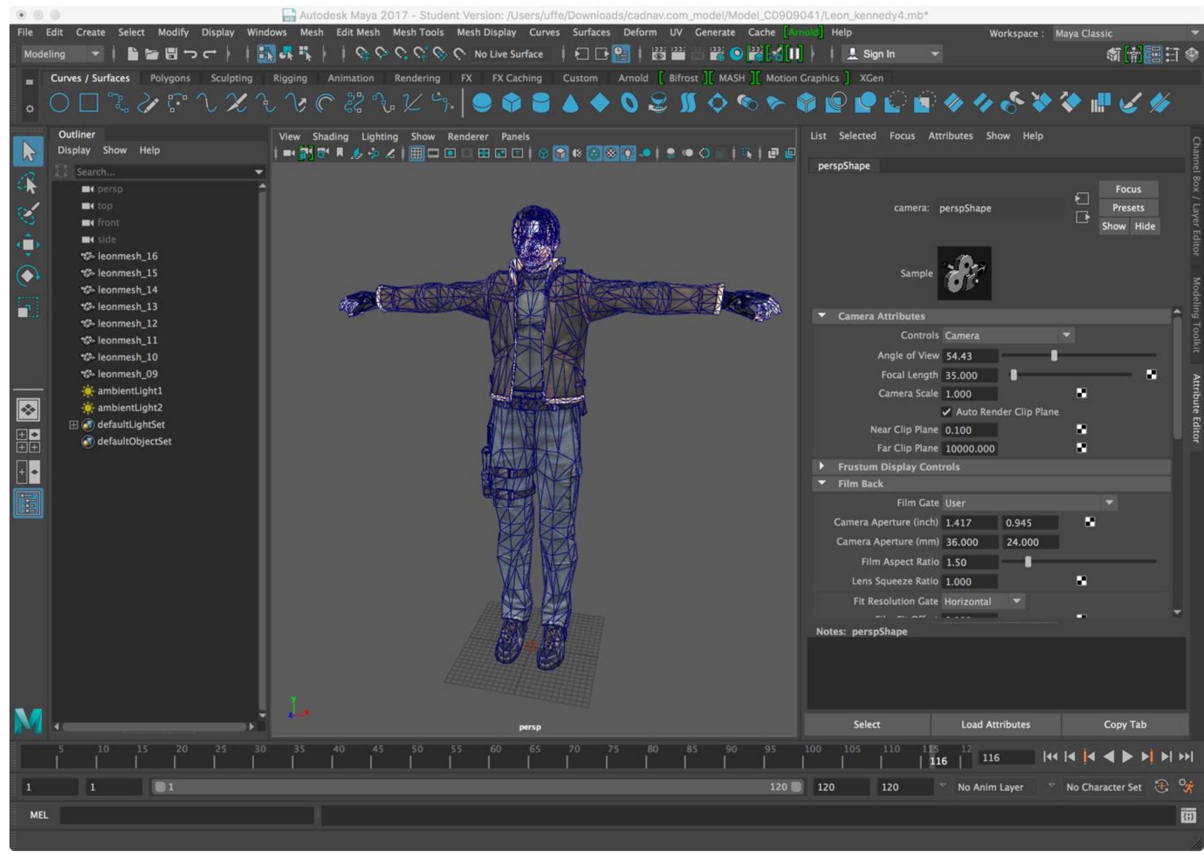
Real-Time Rendering
based on Rasterization

Overview of the Graphics Rendering Pipeline and OpenGL

3D-models: surfaces are constructed by triangles.







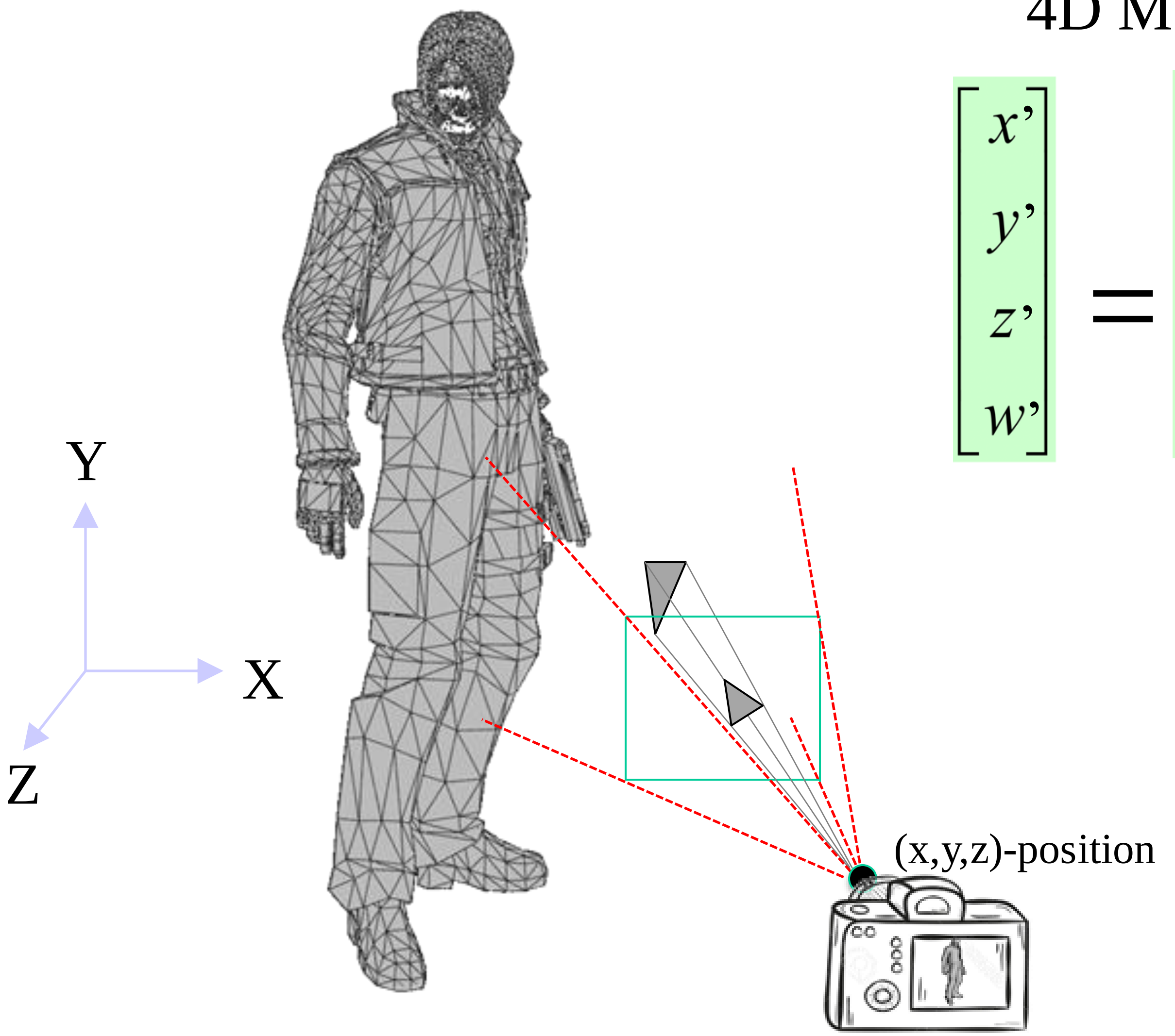
4926 triangles

Why triangles?

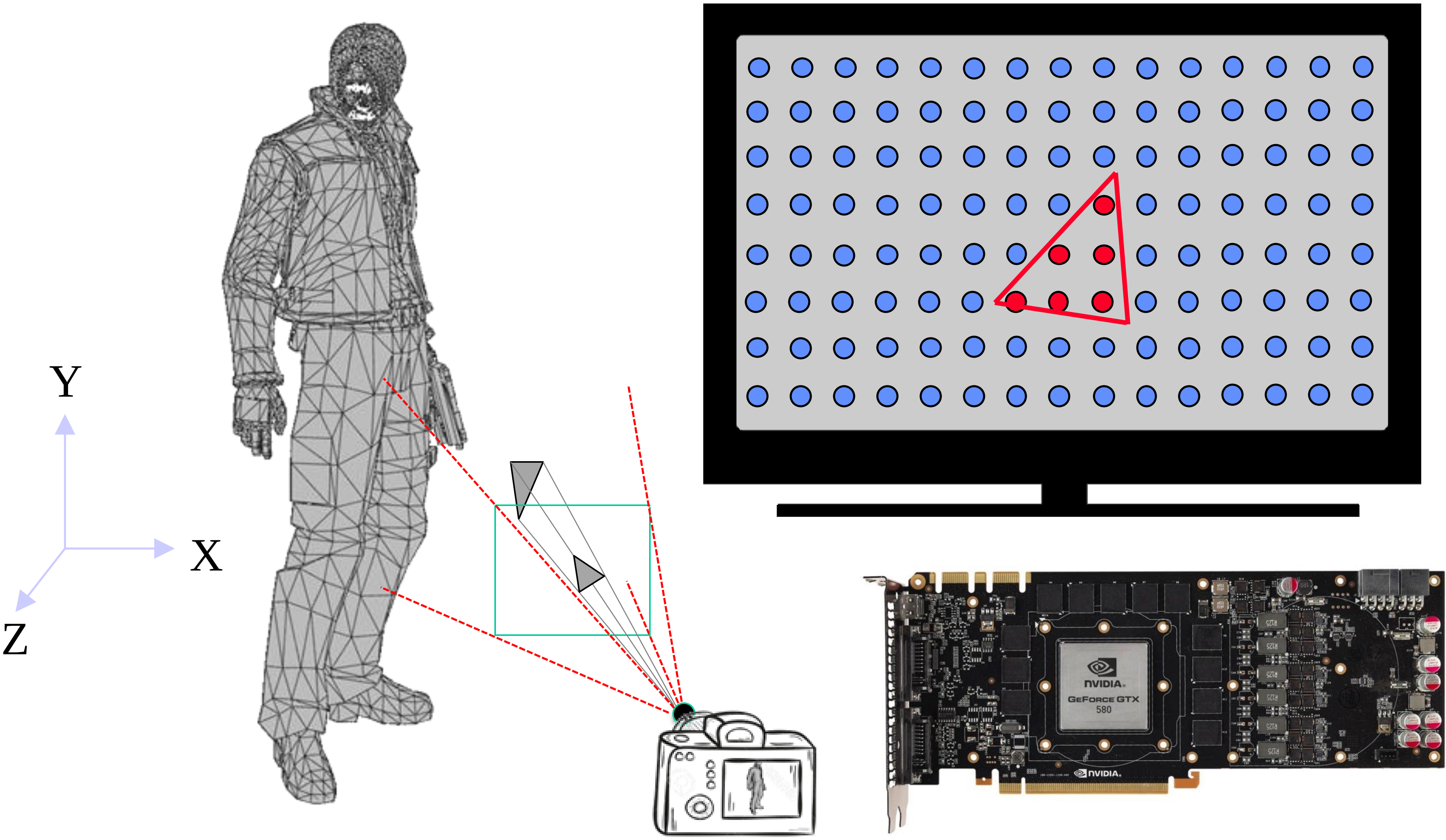
Each triangle is projected onto the image plane using a virtual camera.

4D Matrix Multiplication

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} s_x & \bullet & \bullet & t_x \\ \bullet & s_y & \bullet & t_y \\ \bullet & \bullet & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

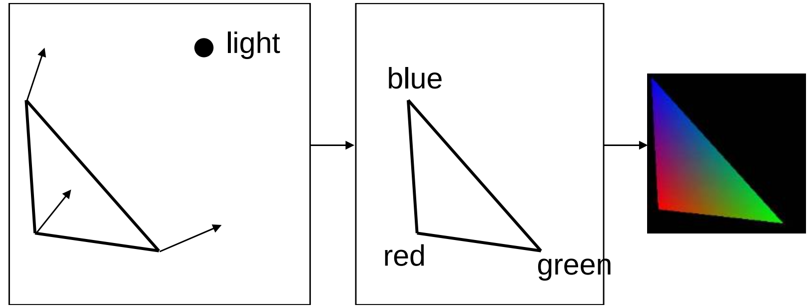


The graphics card draws the triangles onto the screen.

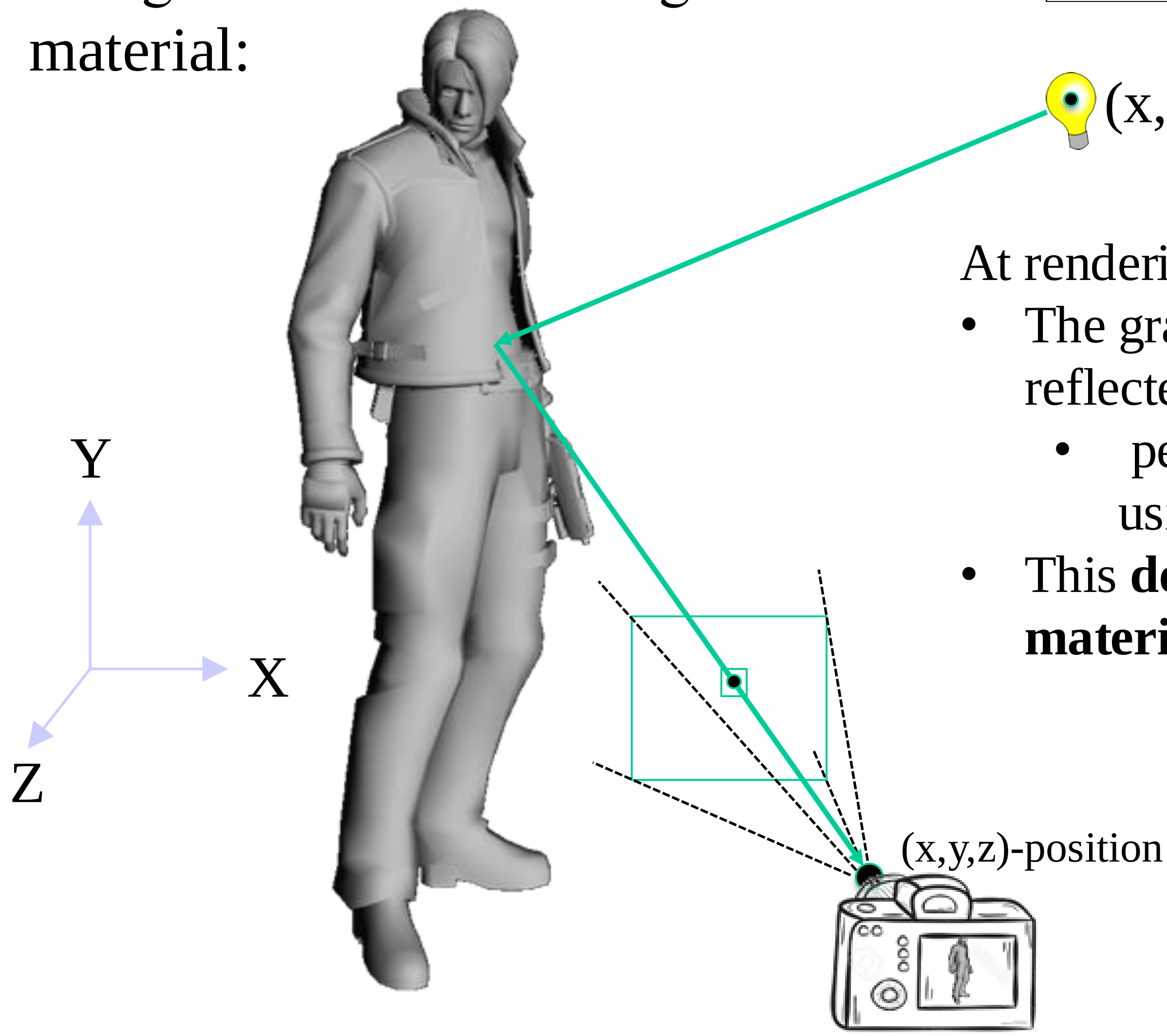


How compute pixel color?

Use some *shading* model based on light sources and triangle's material:



Exaggerated example

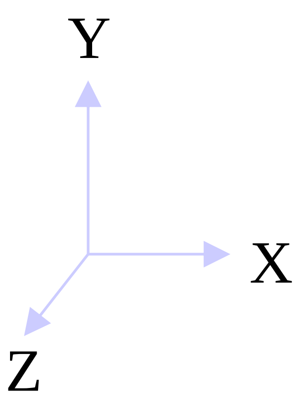


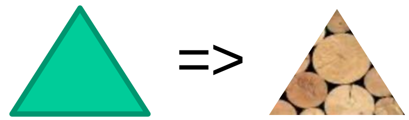
At rendering (for each frame):

- The graphics card computes, the reflected light toward the camera.
  - per pixel (or per vertex and using interpolation per pixel),
- This **depends on the light and material parameters**.

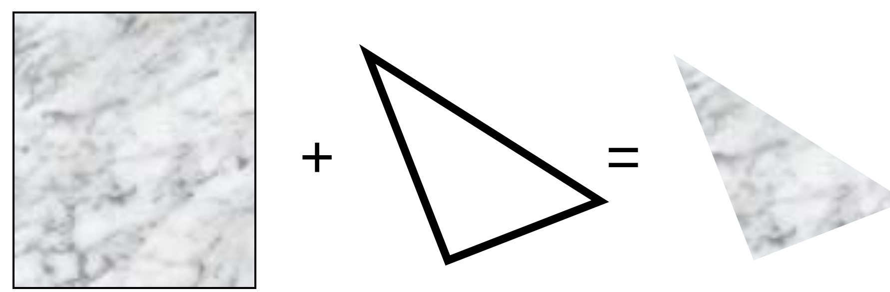
Triangle colors:

are typically multiplied with the lighting contribution. Instead of one single color per triangle, you can use a *texture* (=image) – to simulate details and materials.



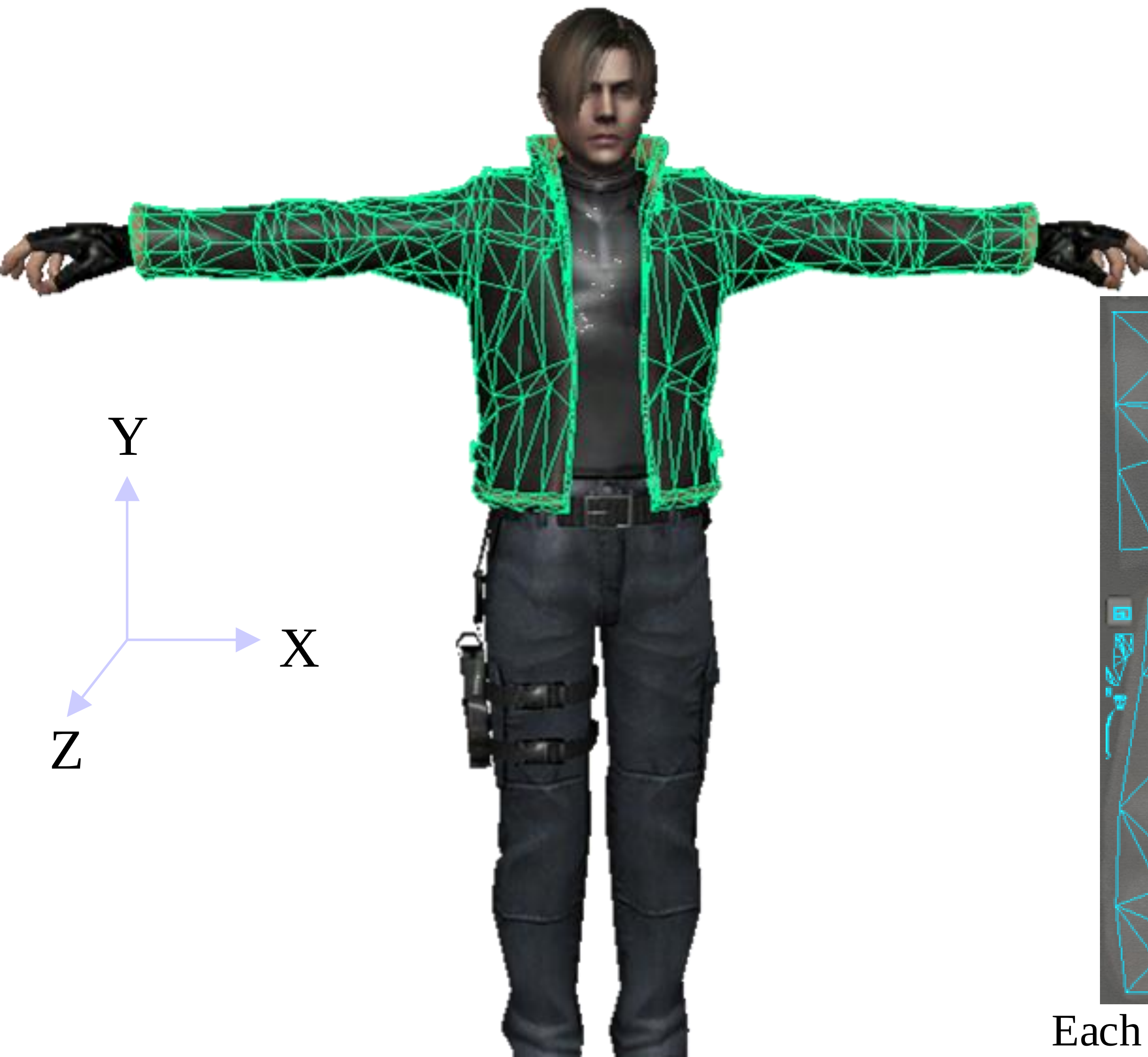


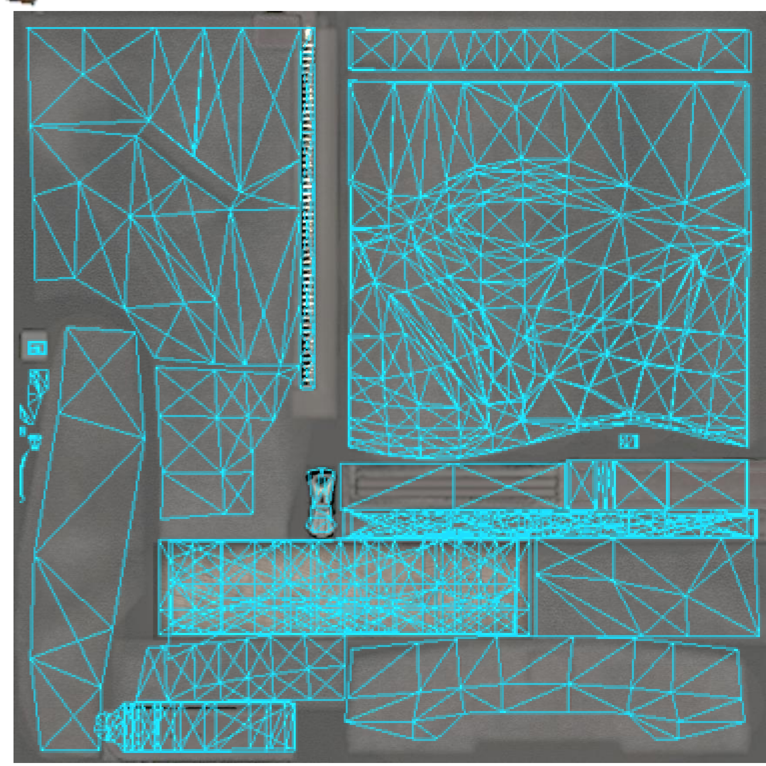
- The **texture** color is modulated (often just multiplied) with the light intensity to get the final pixel color.



Specify which part of the texture that each triangle covers.

Texture Maps

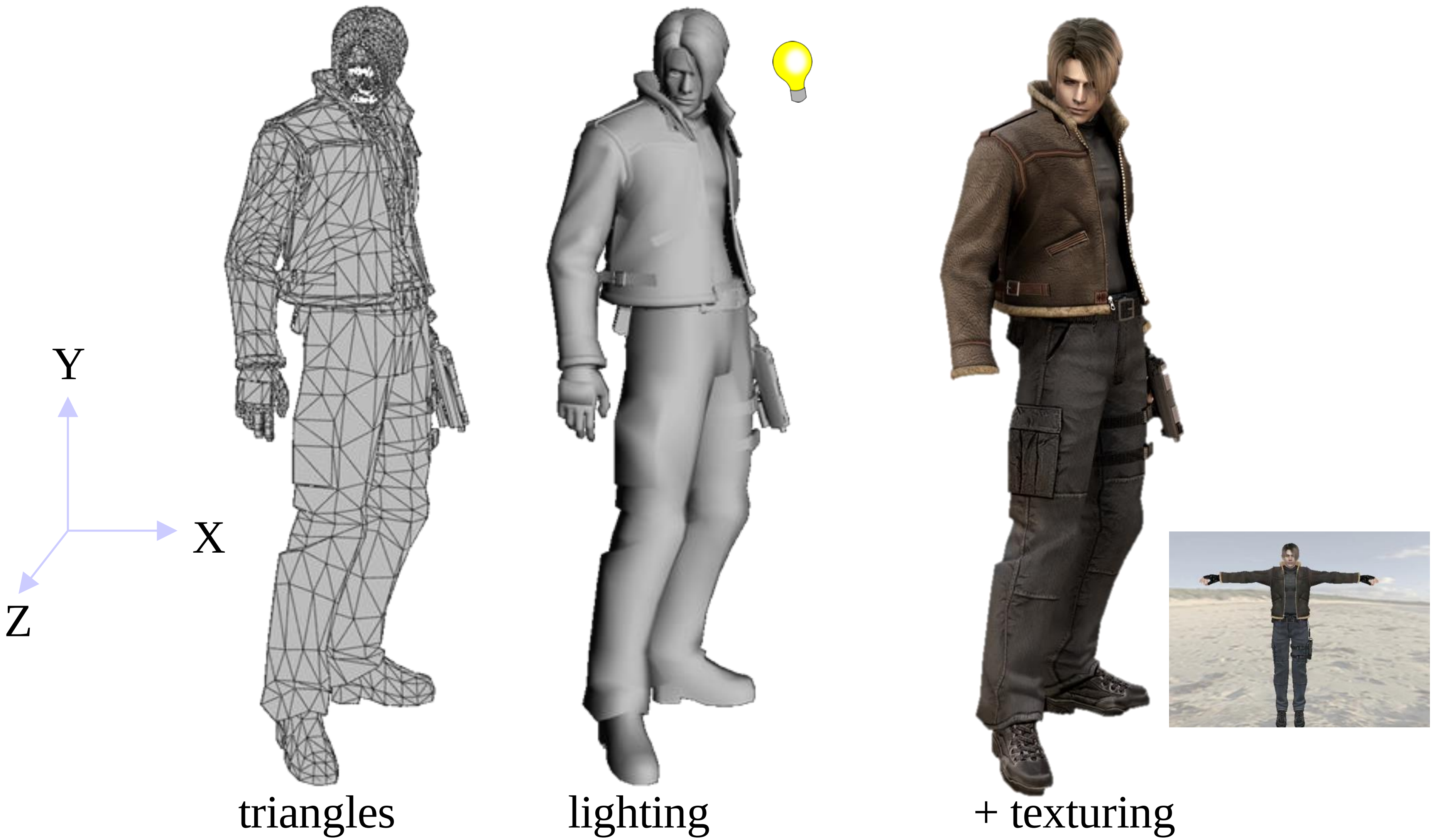




Each triangle’s mapping to texture space

Summary of this very simple type of shading model:

There are many others. Details are given in Lecture 3+4.

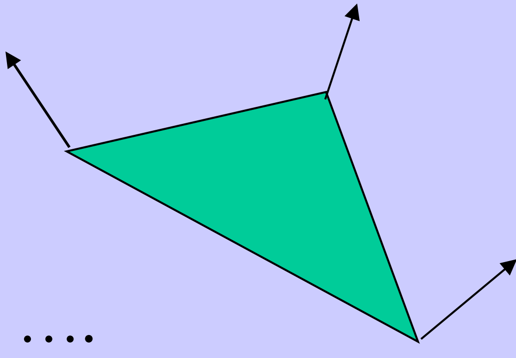


The Graphics Rendering Pipeline

The Application stage, geometry stage, and rasterizer stage

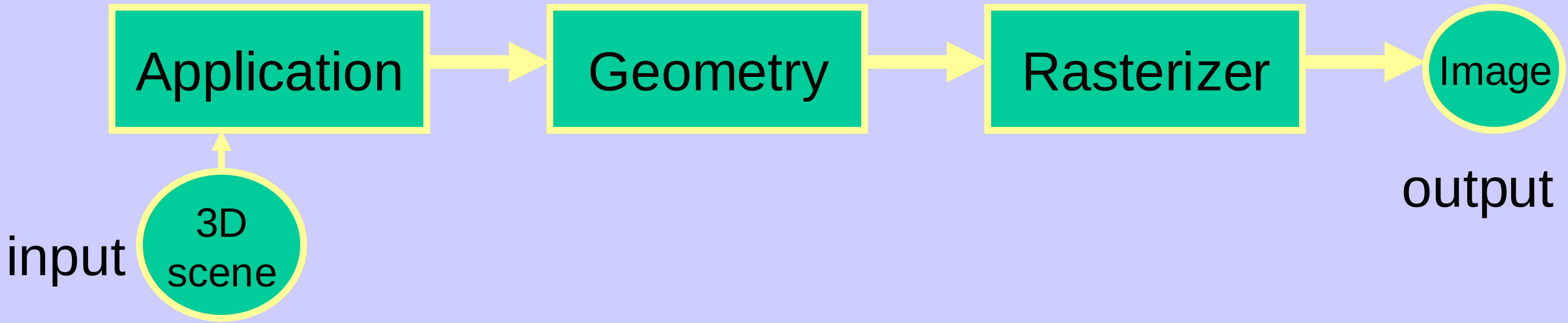
You say that you render a *"3D scene"*, but what is it?

- First, of all to take a picture, it takes a camera – a virtual one.
  - Decides what should end up in the final image
- A 3D scene is:
  - Geometry (triangles, lines, points, and more)
  - Light sources
  - Material properties of geometry
    - Colors, shader code ,
    - Textures (images to glue onto the geometry)
- A triangle consists of 3 vertices
  - A vertex is 3D position, and may have an attached normal, color, texture coordinate, ….



Lecture 1: Real-time Rendering
The Graphics Rendering Pipeline

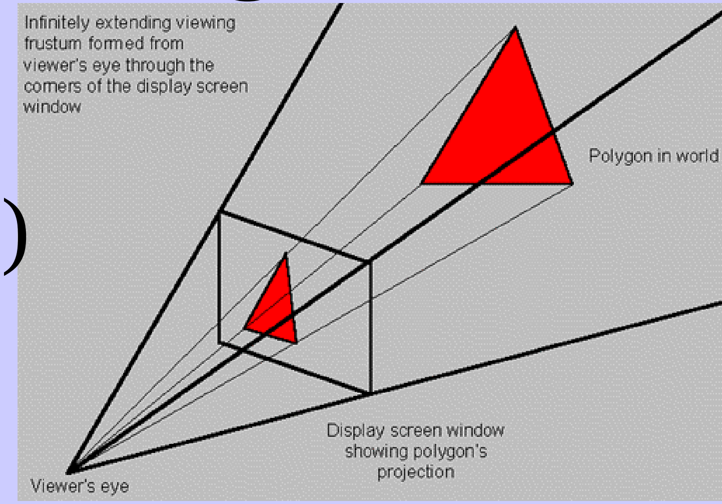
- The pipeline is the ”engine” that creates images from 3D scenes
- Three conceptual stages of the pipeline:
  - Application (executed on the CPU)
  - Geometry
  - Rasterizer



The APPLICATION stage

- Executed on the CPU
  - Means that the programmer decides what happens here
- Examples:
  - Collision detection
  - Speed-up techniques
  - Animation
- Most important task: feed geometry stage with the primitives (e.g. triangles) to render

The GEOMETRY stage

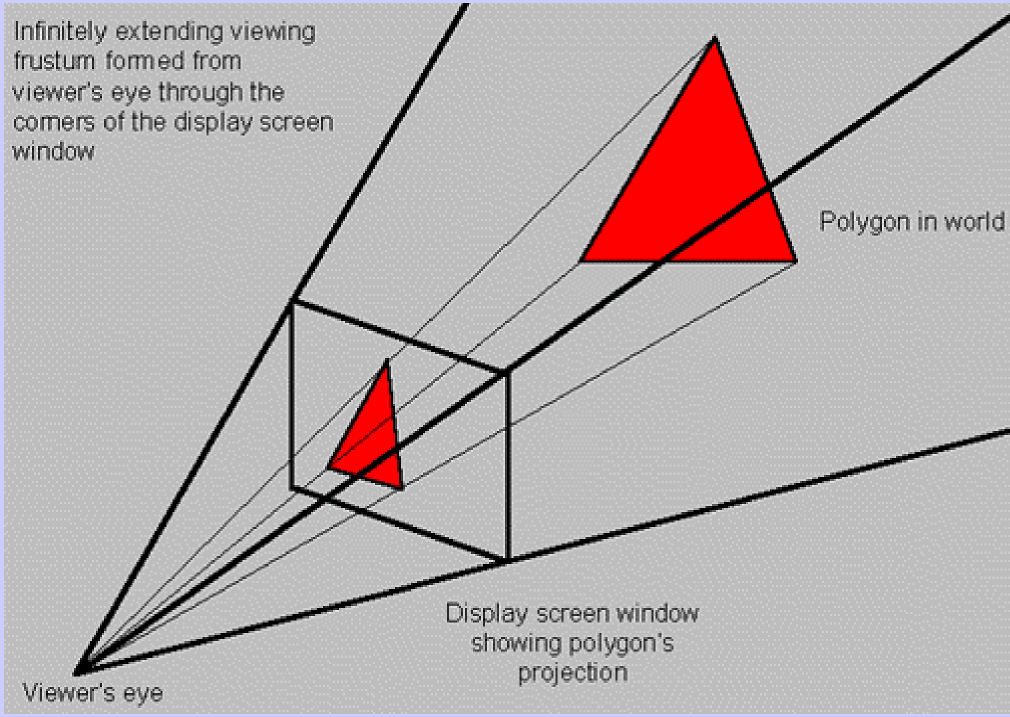


- Task: ”geometrical” operations on the input data (e.g. triangles)
- Allows:
  - Move objects (matrix multiplication)
  - Move the camera (matrix multiplication)
  - Lighting computations per triangle vertex
  - Project onto screen (3D to 2D)
  - Clipping (avoid triangles outside screen)
  - Map to window

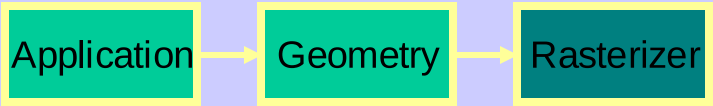
Application → Geometry → Rasterizer

The GEOMETRY stage

Vertex Shading → Projection → Clipping → Screen Mapping

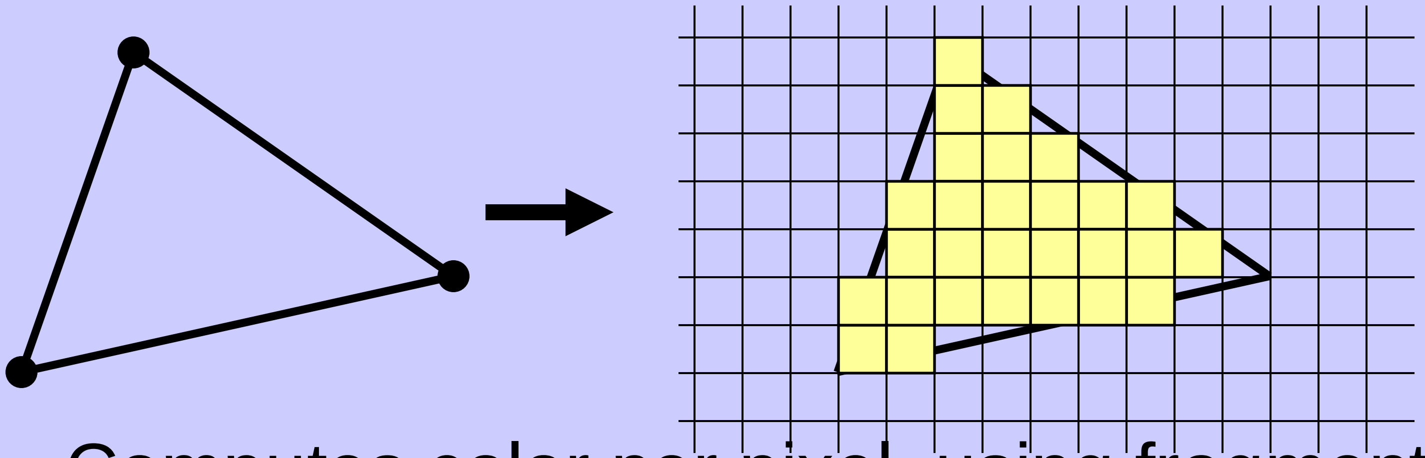


- Vertex Shader
  - A program executed per triangle vertex, computing:
    - Transformations
    - Projection (3D->2D)
    - Compute a color per vertex
- Clipping
- Screen Mapping



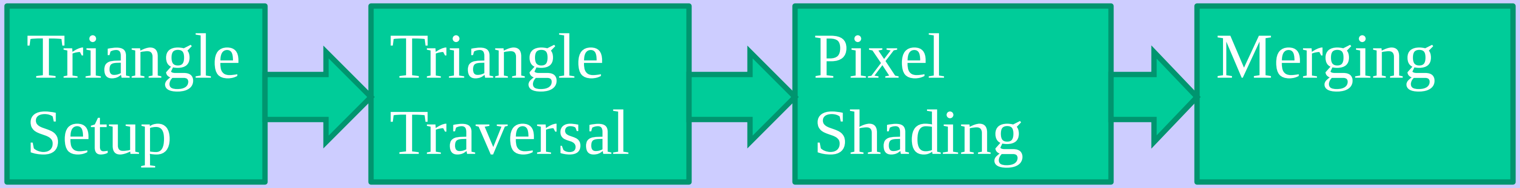
The RASTERIZER stage

- Main task: take output from GEOMETRY and turn into visible pixels on screen



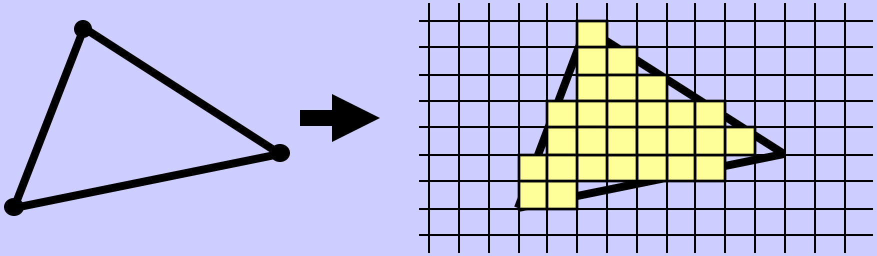
- Computes color per pixel, using fragment shader (=pixel shader)
  - textures, (light sources, normal), colors and various other per-pixel operations
- And visibility is resolved here using the fragment's z-value to check its visibility

The rasterizer stage



Triangle Setup:

- collect three vertices + interpolated vertex shader output (incl. normals) and make one triangle.



Triangle Traversal

- Scan conversion or rasterization

Pixel Shading

- Compute pixel color

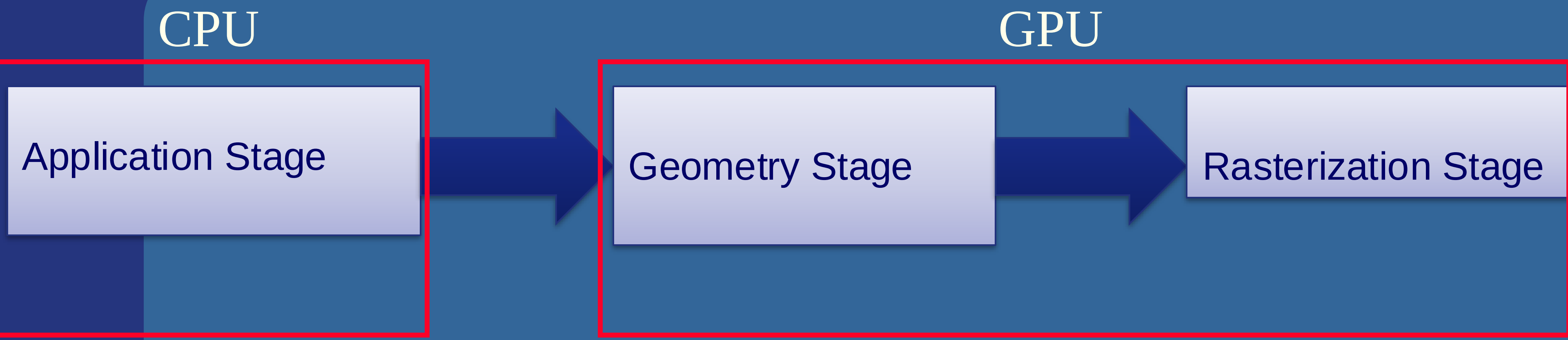
Merging:

- output color to screen

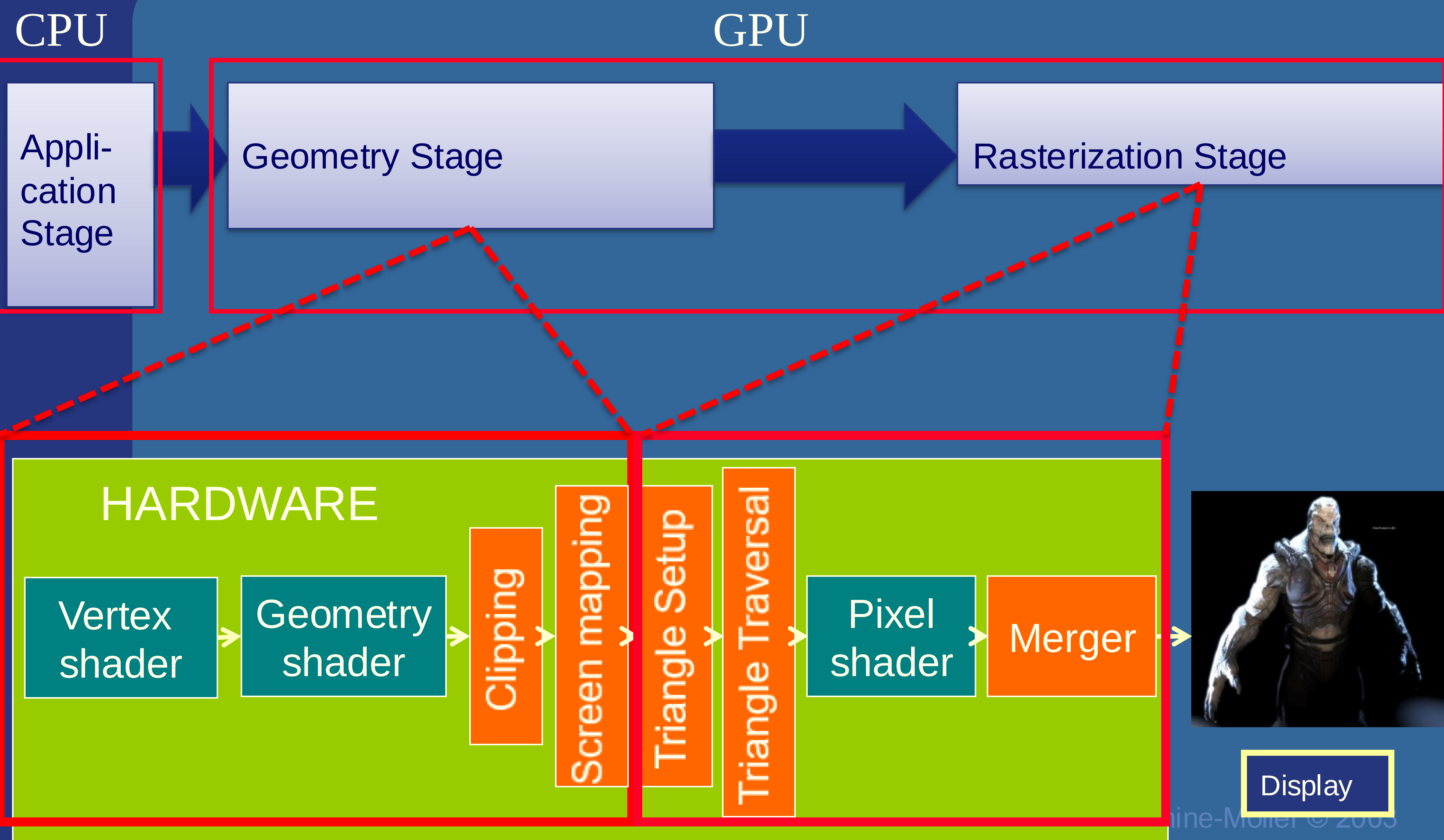
The three stages' correlation to hardware

The Application stage, geometry stage, and rasterizer stage

Rendering Pipeline and Hardware



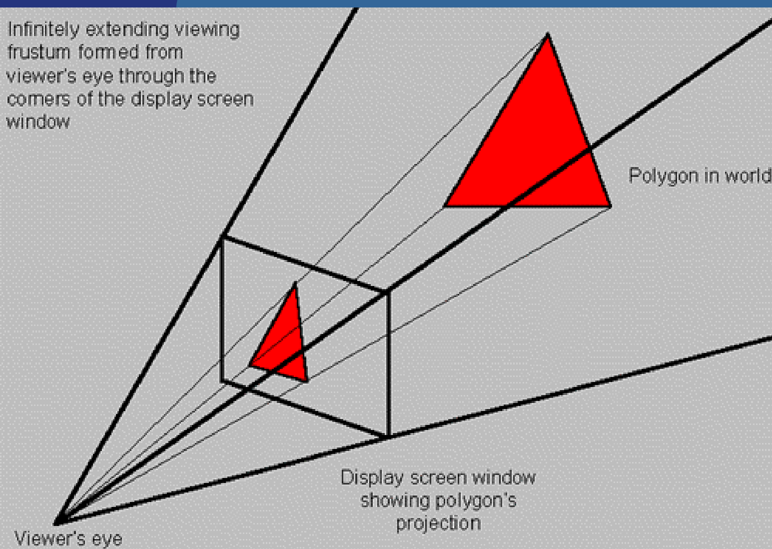
Rendering Pipeline and Hardware

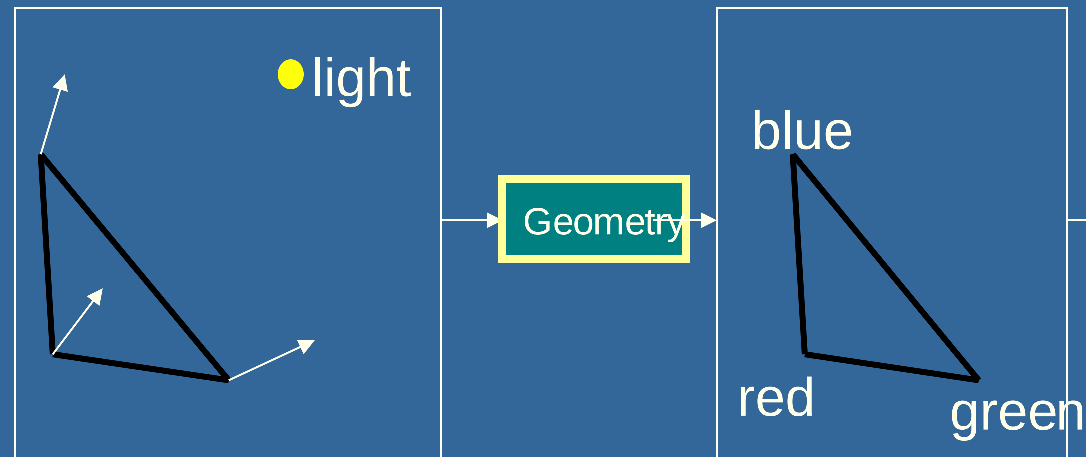


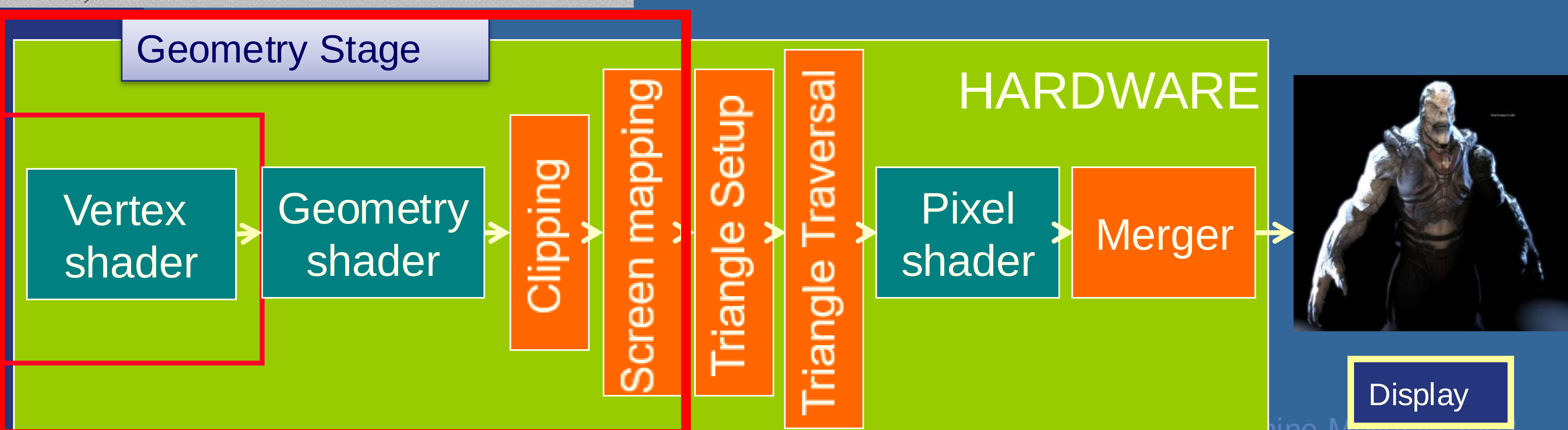
Hardware design

Vertex shader:

- Lighting (colors)
- Screen space positions



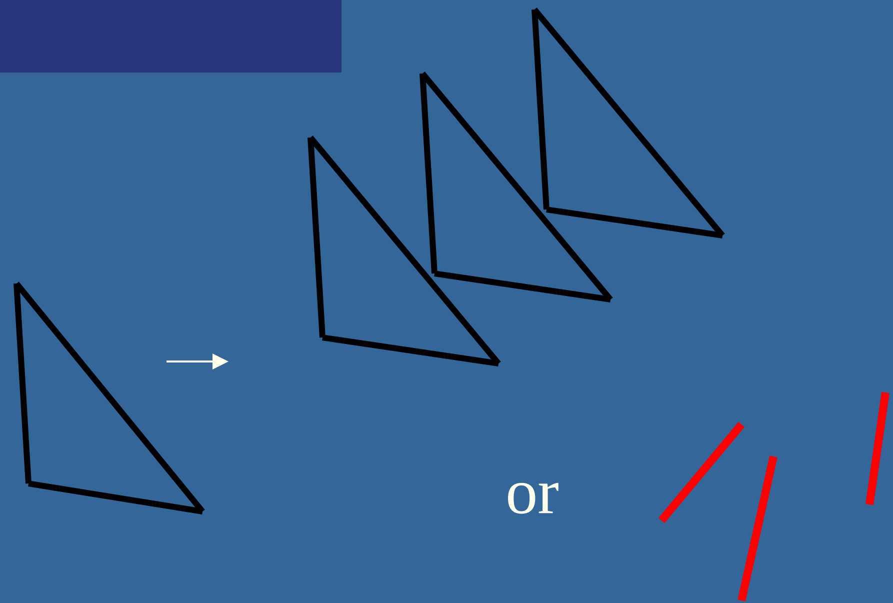


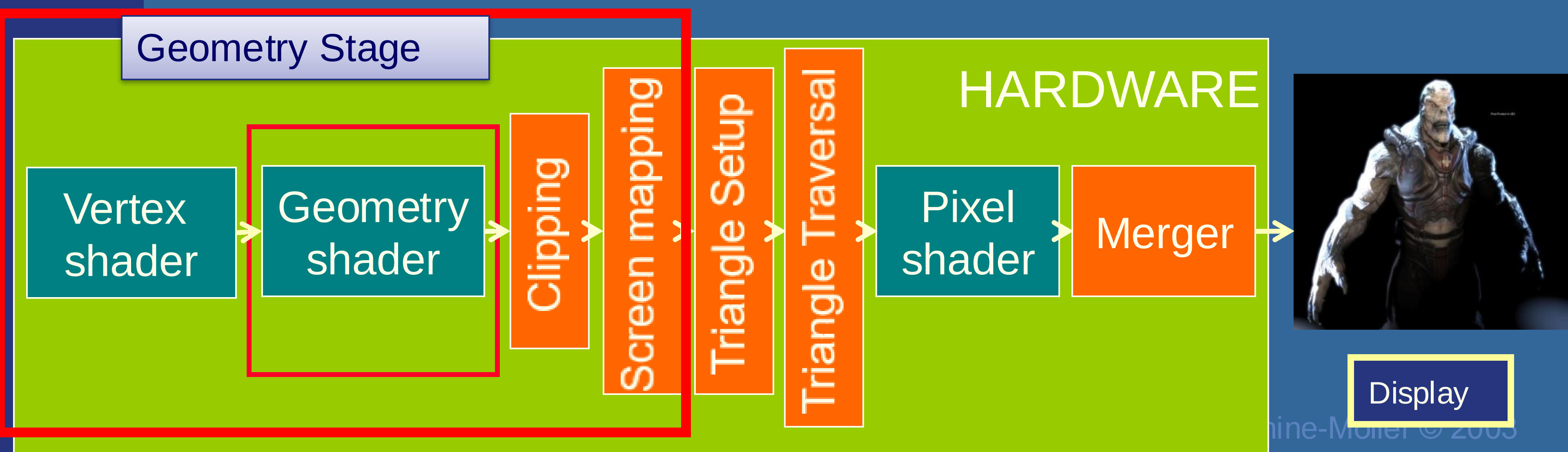


Hardware design

Geometry shader:

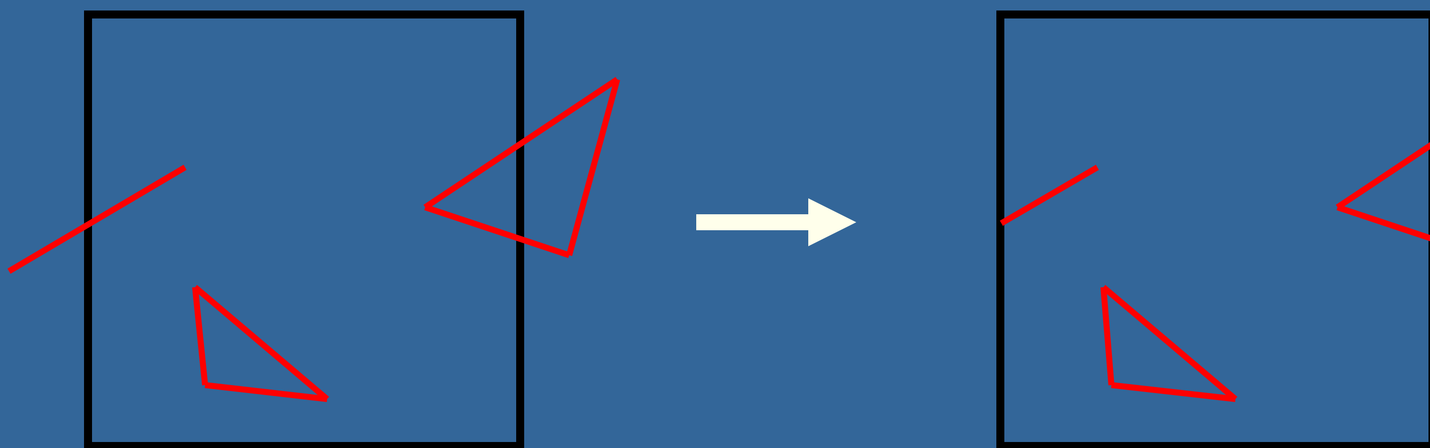
- One input primitive
- Many output primitives

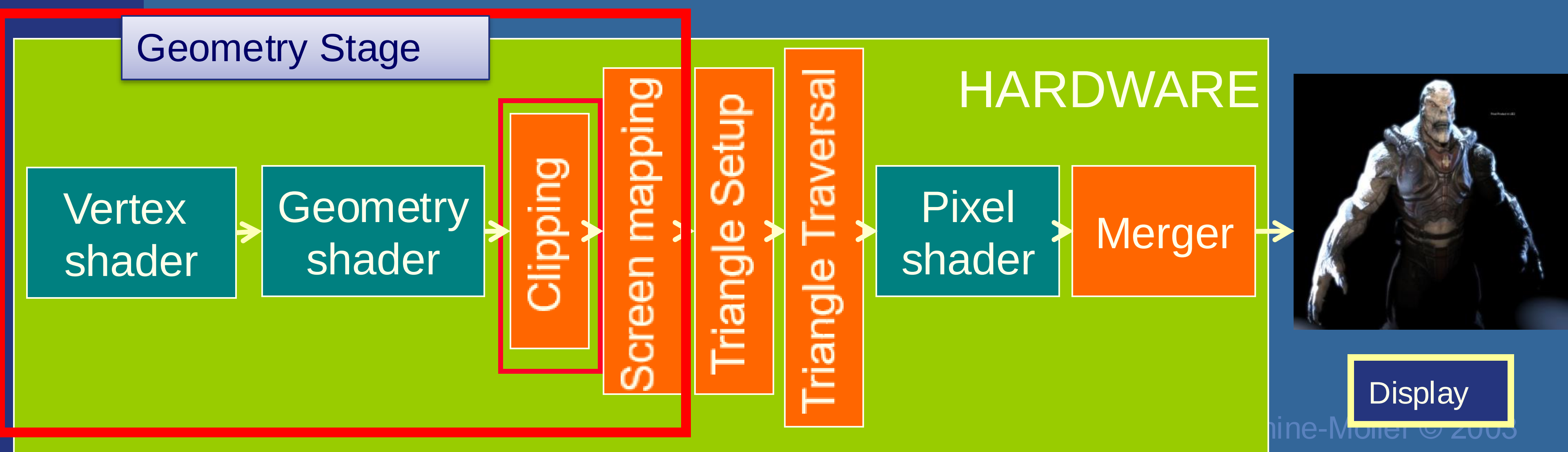




Hardware design

Clips triangles against the unit cube (i.e., ”screen borders”)

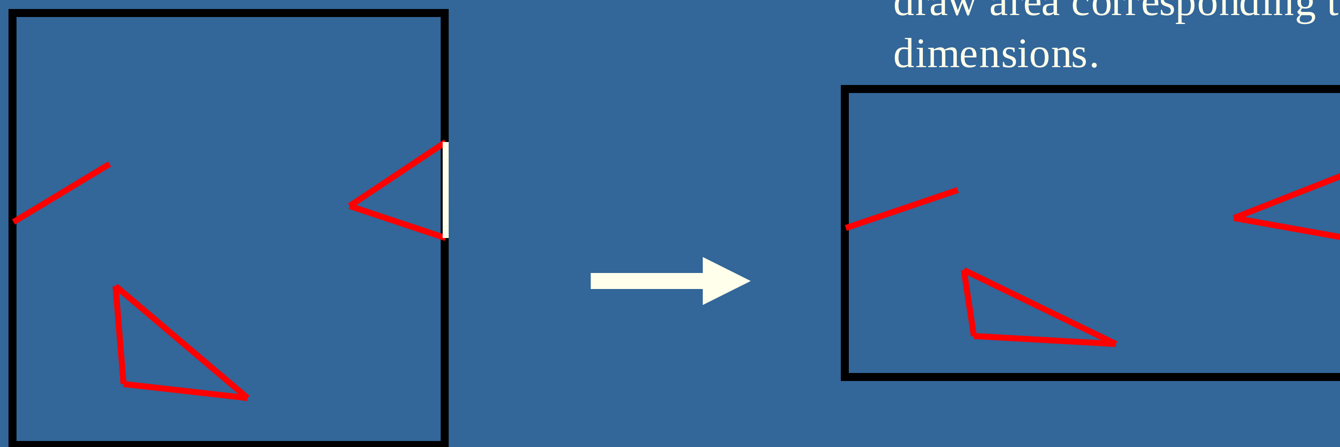


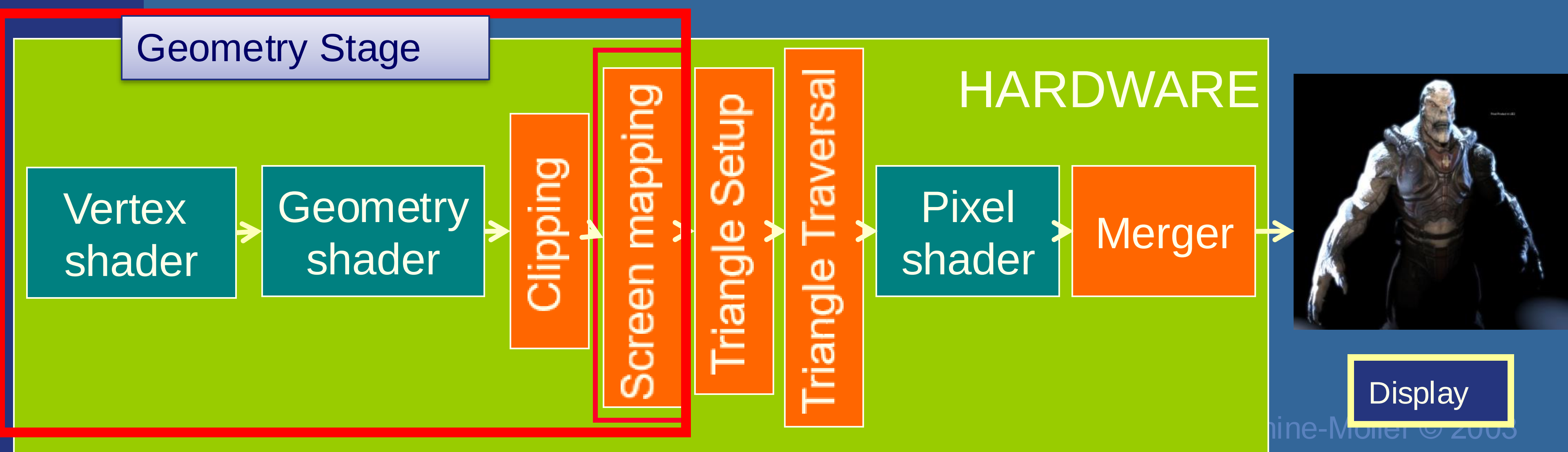


Hardware design

Maps window size to unit cube

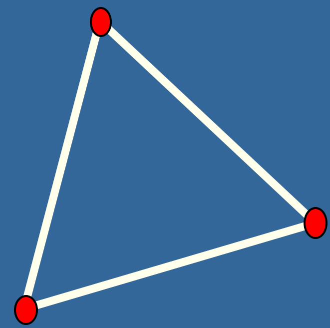
Geometry stage always operates inside a unit cube [-1,-1,-1]-[1,1,1]
Next, the rasterization is made against a draw area corresponding to window dimensions.

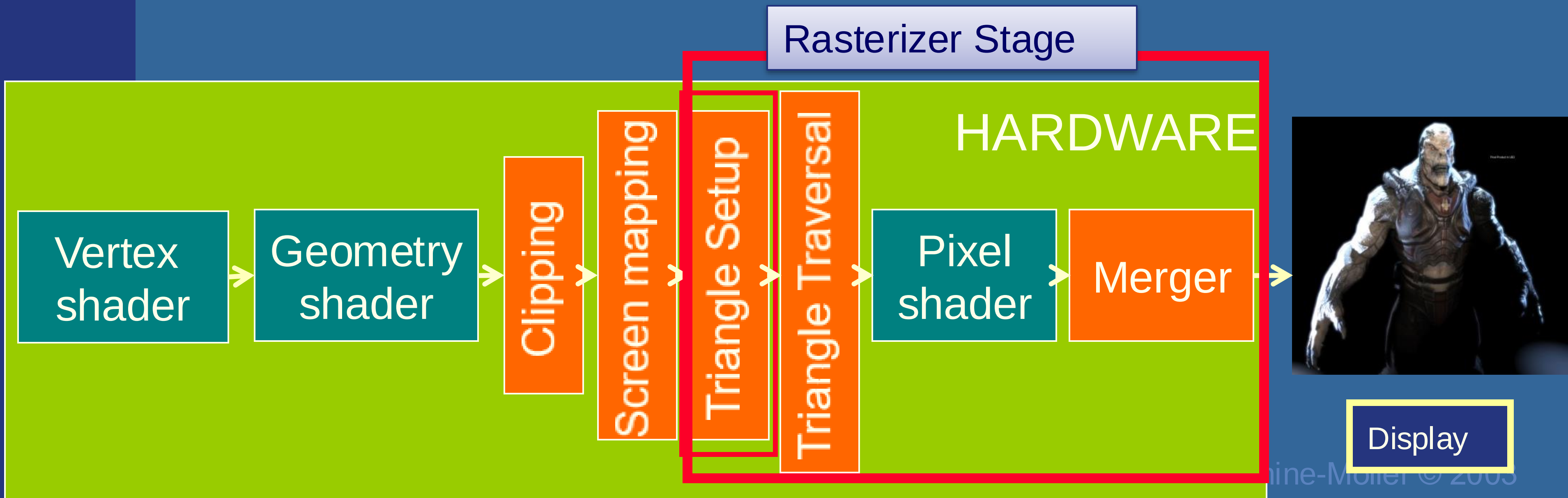




Hardware design

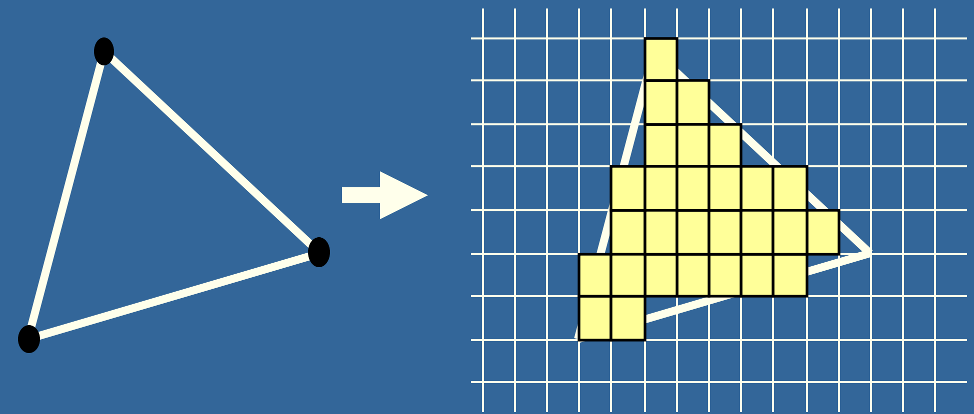
Collects three vertices into one triangle

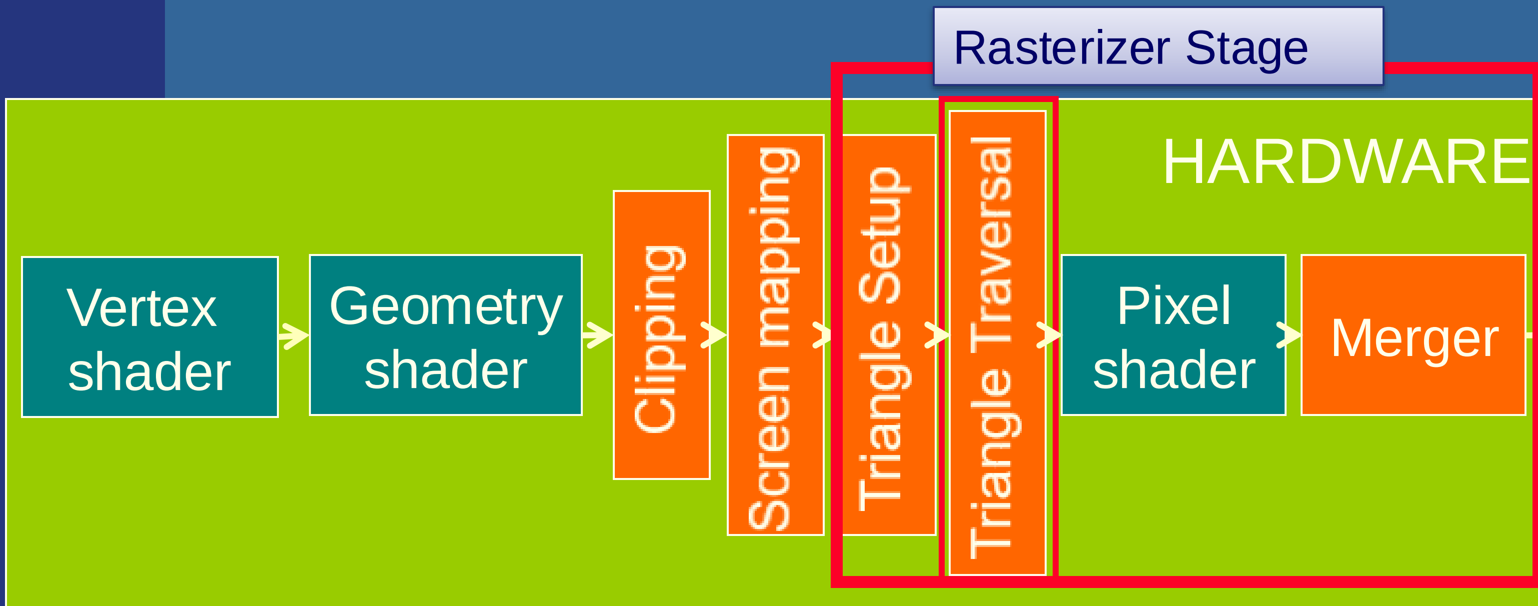




Hardware design

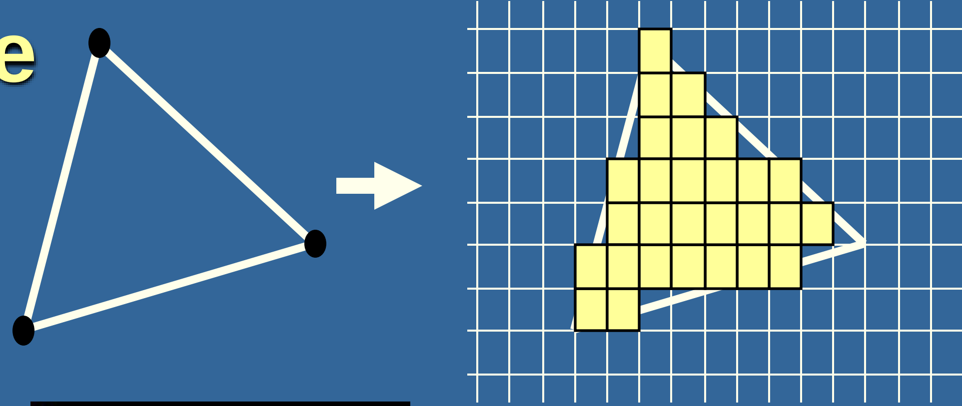
Creates the fragments/pixels for the triangle

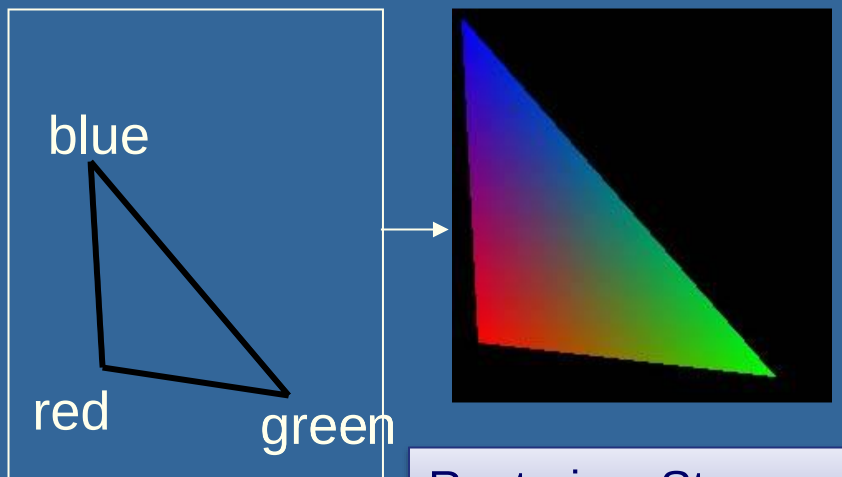




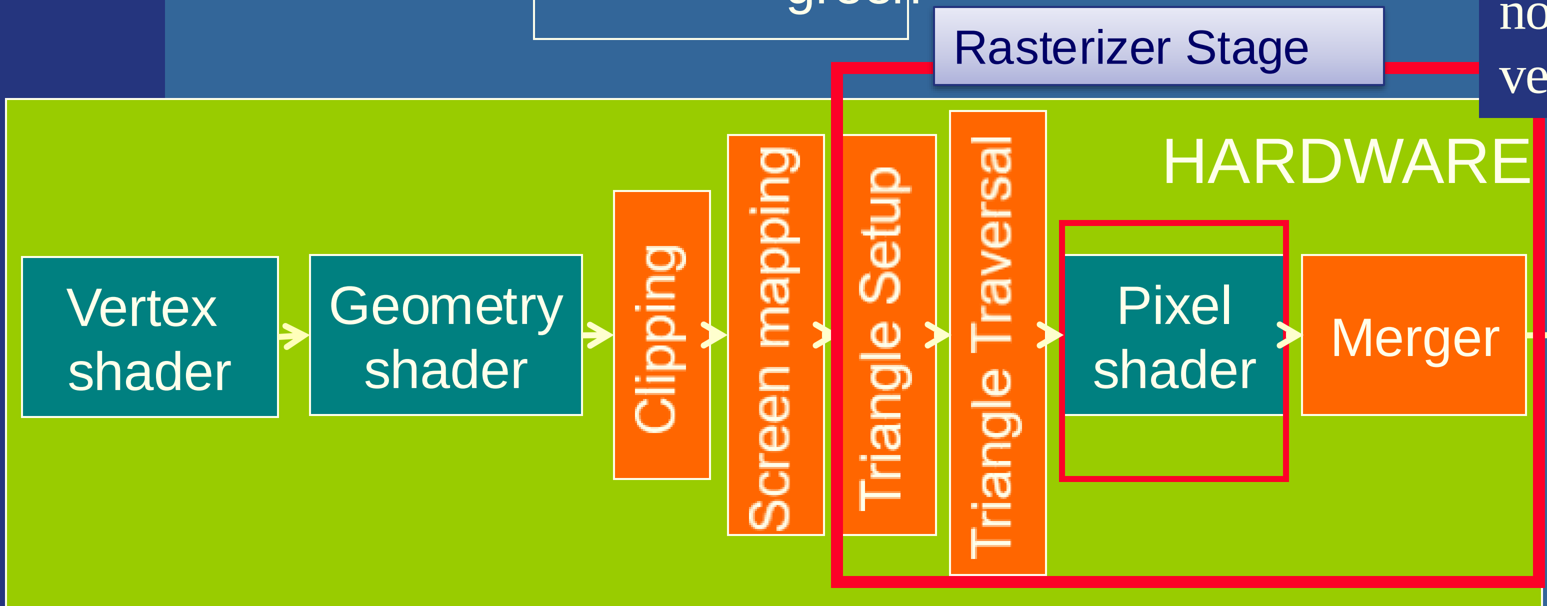
Display

Graphics Hardware





Pixel Shader:
Compute color using:
•Textures
•Interpolated data (e.g. Colors + normals) from vertex shader



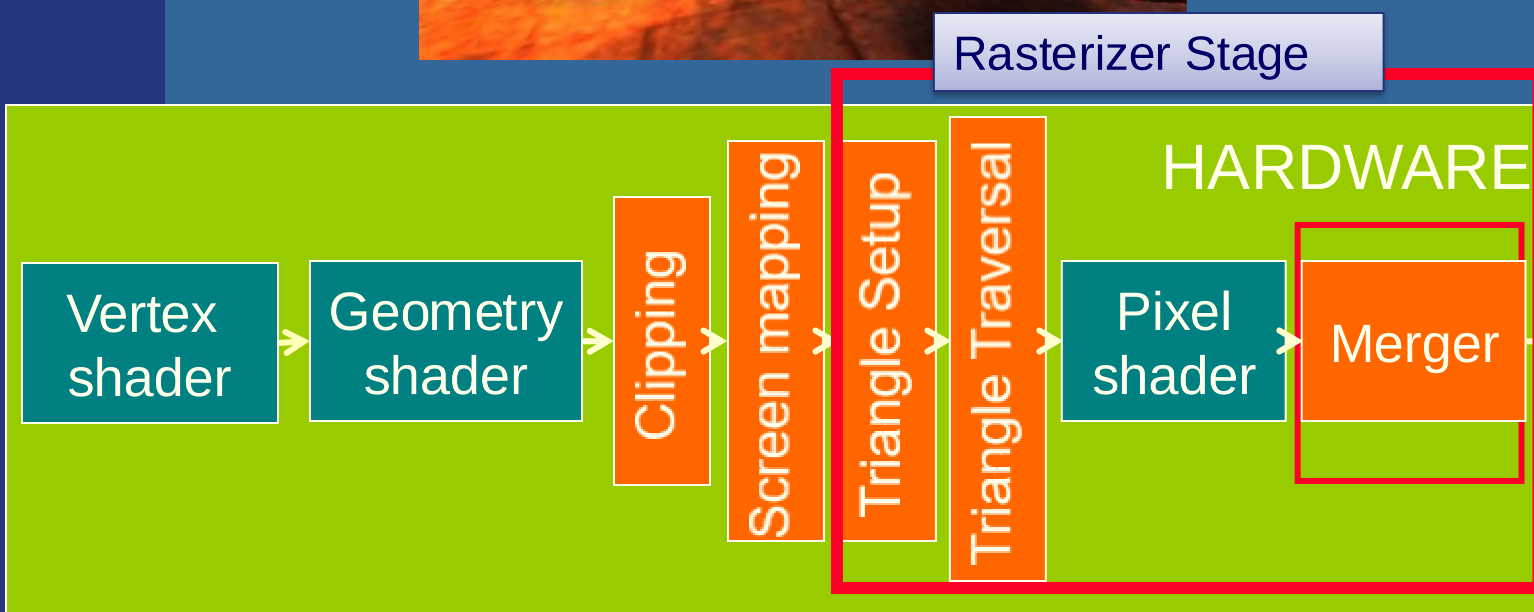
Display

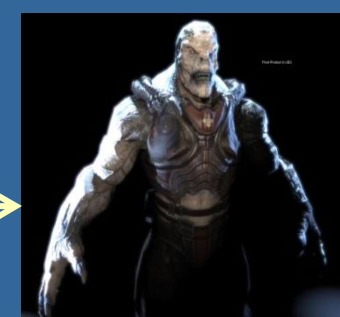
Hardware design

The merge units update the frame buffer with the pixel's color

Frame buffer:

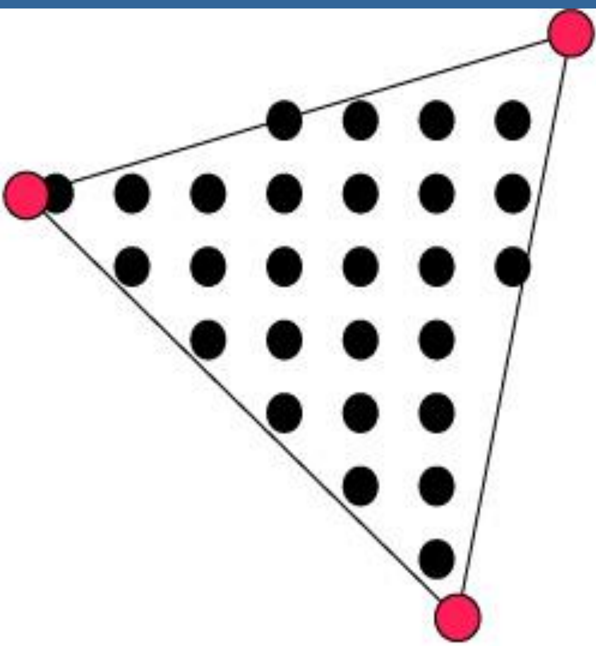
- Color buffers
- Depth buffer
- Stencil buffer





Display

What are vertex and fragment (pixel) shaders?

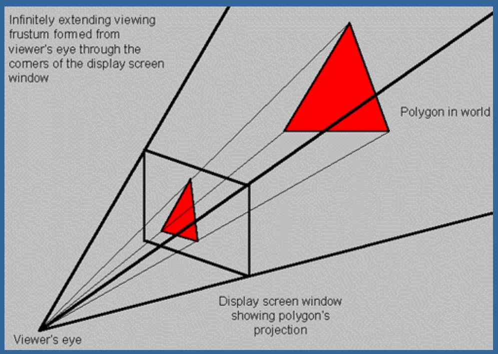


- Vertex shader: reads from textures. Writes outputs per vertex, which are interpolated and input to the fragment shader invocation per pixel.
- Fragment shader: reads from textures, writes to pixel color.
- Memory: Texture memory (read + write) typically 4 GB – 16 GB
- Program size: the smaller the faster

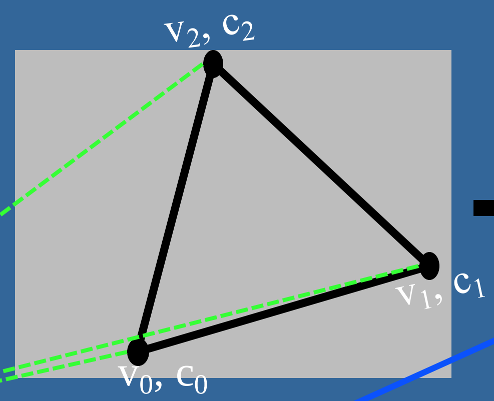
● For each vertex, a vertex program (vertex shader) is executed

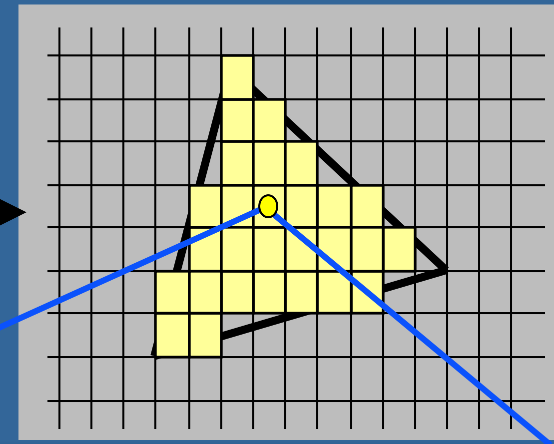
● For each fragment (pixel) a fragment program (fragment shader) is executed

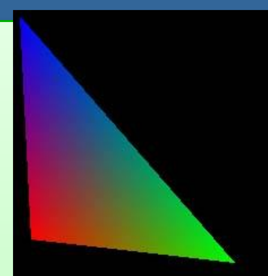
Shaders



v = vertex (x,y,z)
c = color (r,g,b)







```
// Vertex Shader, executed per vertex
#version 420

layout(location = 0) in vec3 vertex; // x,y,z
layout(location = 1) in vec3 color; // r,g,b
out vec3 vsOutColor; // output result for this vertex
uniform mat4 modelViewProjectionMatrix;

void main()
{
    gl_Position = modelViewProjectionMatrix *
            vec4(vertex,1); //Project the vertex from 3D
    vsOutColor = color; // Output for this vertex. Will
                // be interpolated input to
                // fragment shader
}
```

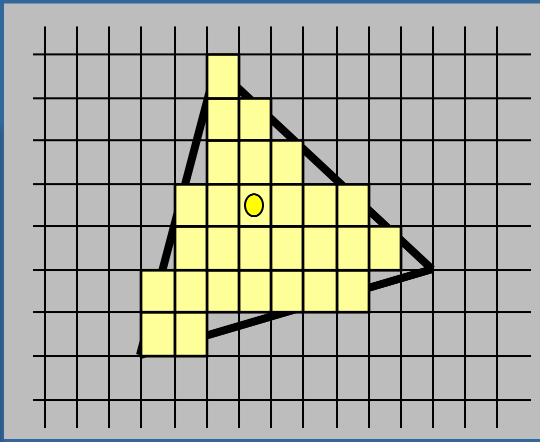
```
// Fragment Shader, exec. per fragment
#version 420
precision highp float; // 32-bits floats

in  vec3 vsOutColor; // interpolated input
                       from vertex shader
layout(location = 0) out vec4 fragColor;
    // Here, location=0 means that we  draw
    to frameBuffer[0], i.e., the screen

void main()
{
    fragColor = vec4(vsOutColor,1);
}
```

Shaders

Example of a more advanced fragment shader:

```
precision highp float;

uniform sampler2D tex0;
uniform sampler2D tex1;
uniform sampler2D tex2;
uniform sampler2D tex3;

uniform float val;

varying vec2 uv_0;
varying vec3 n;

void main(void) {
        gl_FragColor.rgb = compute_color();
        gl_FragColor.a = 1.0;
}
```

```
vec3 compute_color()
{
    vec4 gbuffer = texture2D(tex0, uv_0);
        int intColor = int(gbuffer.x);
        int r = (intColor/256)/256;
        intColor -= r*256*256;
        int g = intColor/256;
        intColor -= g*256;
        int b = intColor;
         vec3 color = vec3(float(r)/255.0, float(g)/255.0,
        float(b)/255.0 );

        normal = vec3(sin(gbuffer.g) * cos(gbuffer.b),
        sin(gbuffer.g)*sin(gbuffer.b), cos(gbuffer.g));
        vec2 ang = gbuffer.gb*2.0-vec2(1.0);
        vec2 scth = vec2( sin(ang.x * PI), cos(ang.x * PI);
        vec2 scphi = vec2(sqrt(1.0 - ang.y*ang.y), ang.y);
        normal = -vec3(scth.y*scphi.x, scth.x*scphi.x, scphi.y);
        roughness = 0.05;
        specularity = 1.0;
        fresnelR0 = 0.3;
        return color;
}
```



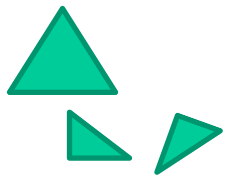
OpenGL
(Open Graphics Library)

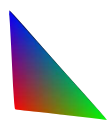
CPU-side

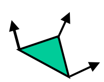
Language: C++
API: OpenGL (Direct3D)
Window system: SDL (Cocoa, Win32,…)

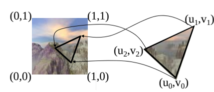
C++:

```
float positions[] = {
// X     Y    Z  per vertex
  0.0f,  0.5f, 1.0f, // v0
 -0.5f, -0.5f, 1.0f, // v1
  0.5f, -0.5f, 1.0f  // v2
  ...                          // ... vn
};
// and any other per-vertex data, e.g.:
float colors[] = {
// R     G    B
  1.0f,  0.0f, 0.0f, // c0
 -0.0f,  1.0f, 0.0f, // c1
  0.0f,  0.0f, 1.0f  // c2
  ...                          // ... cn
};
float normals[] = {...};
float textureCoords[] = {...};
```









OpenGL C++ API:

Vertex-buffer objects

```
uint32 positionBuffer; // x,y,z per vertex
uint32 colorBuffer;    // r,g,b per vertex
```

Vertex-Array object // groups the arrays

```
uint32 vertexArrayObject;
```

Shaders

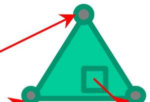
```
uint32 vertexShader;
uint32 fragmentShader;
uint32 shaderProgram;
```

GPU-side

Language: GLSL
used for vertex- ,geometry-, and fragment shaders

Vertex Shader:

```
#version 420

layout(location = 0) in  vec3 position;
layout(location = 1) in  vec3 color;

out vec3 outColor; // r,g,b

void main()
{
   gl_Position = vec4(position, 1.0);
   outColor = color;
}
```



Fragment Shader:

```
#version 420

precision highp float; // required by GLSL spec Sect 4.5.3
                       // (though nvidia does not, amd does)

layout(location = 0) out vec4 fragmentColor;
in vec3 outColor;

void main()
{
   // fragmentColor = vec4(1,1,1,1);
   fragmentColor.rgb = outColor;
   fragmentColor.a = 1.0;
}
```

Per-pixel-interpolated value

CPU-side

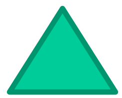
Language: C++

API: OpenGL (Direct3D)

Window system: SDL (Cocoa, Win32,…)

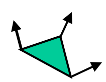
C++:

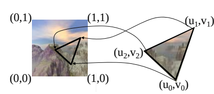
```
float positions[] = {
// X      Y    Z
  0.0f,  0.5f, 1.0f, // v0
 -0.5f, -0.5f, 1.0f, // v1
  0.5f, -0.5f, 1.0f  // v2
```



How to connect the vertexArrayObject as vertex shader input (position, color):

```
  0.0f,  0.0f, 1.0f  // c2
  ...                              // ... cn
};
float normals[] = {…};
float textureCoords[] = {…};
```





OpenGL C++ API:

Vertex-buffer objects

```
uint32 positionBuffer; // x,y,z per vertex
uint32 colorBuffer;    // r,g,b per vertex
```

Vertex-Array object // groups the arrays

```
uint32 vertexArrayObject;
```

Shaders

```
uint32 vertexShader;
uint32 fragmentShader;
uint32 shaderProgram;
```

GPU-side

Language: GLSL

used for vertex- ,geometry-, and fragment shaders

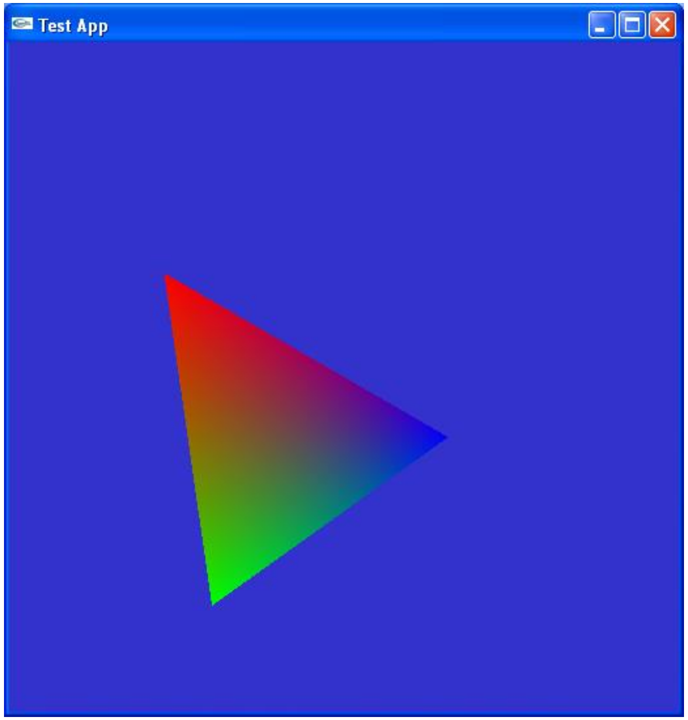
Vertex Shader:

```
#version 420

layout(location = 0) in  vec3 position;
layout(location = 1) in  vec3 color;

out vec3 outColor;

void main()
{
    gl_Position = vec4(position, 1.0);
    outColor = color;
}
```

```
glGenVertexArrays(1, &vertexArrayObject);
// Following commands now affect this vertex array object.
glBindVertexArray(vertexArrayObject);

// Makes positionBuffer the current array buffer for subsequent commands.
// and attaches positionBuffer to vertexArrayObject,
glBindBuffer( GL_ARRAY_BUFFER, positionBuffer );
// Connect positions to location 0. 3 floats per vertex
glVertexAttribPointer(0, 3, GL_FLOAT, …, positions );

// Makes colorBuffer the current array buffer for subsequent commands.
// and attaches colorBuffer to vertexArrayObject,
glBindBuffer( GL_ARRAY_BUFFER, colorBuffer );
// Connect colors to location 1. 3 floats per vertex
glVertexAttribPointer(1, 3, GL_FLOAT, …, colors );

glEnableVertexAttribArray(0); // Enable attribute array 0
glEnableVertexAttribArray(1); // Enable attribute array 1
```

A Simple Program
Computer Graphics version of “Hello World”

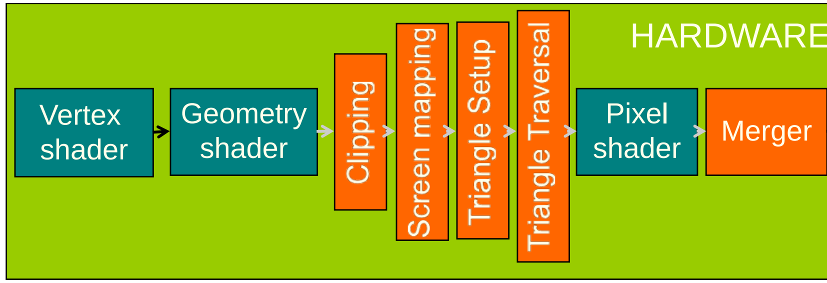
Generate a triangle on a solid background



BONUS

Graphics Pipelines

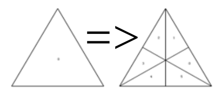
We focus on:

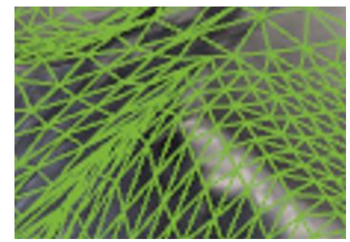


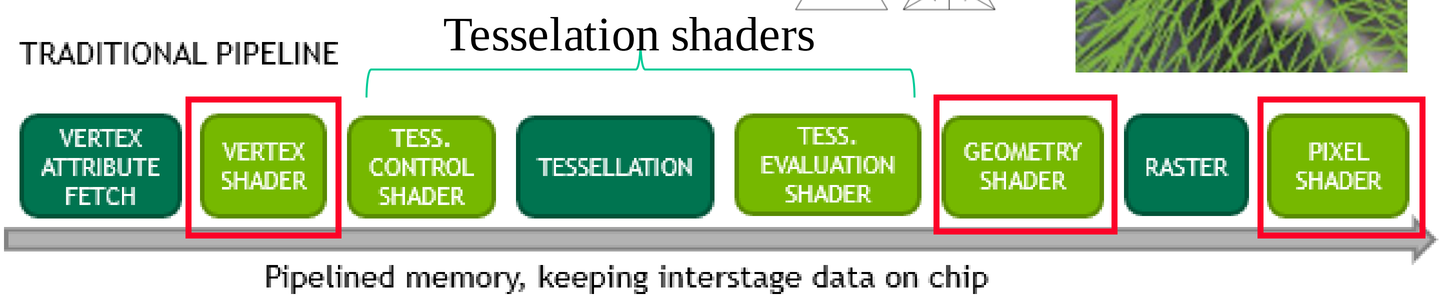
<u>Compatibility</u>:
- OpenGL 4.3
- WebGL 2
- OpenGLES 3

i.e., phones, web, PCs…

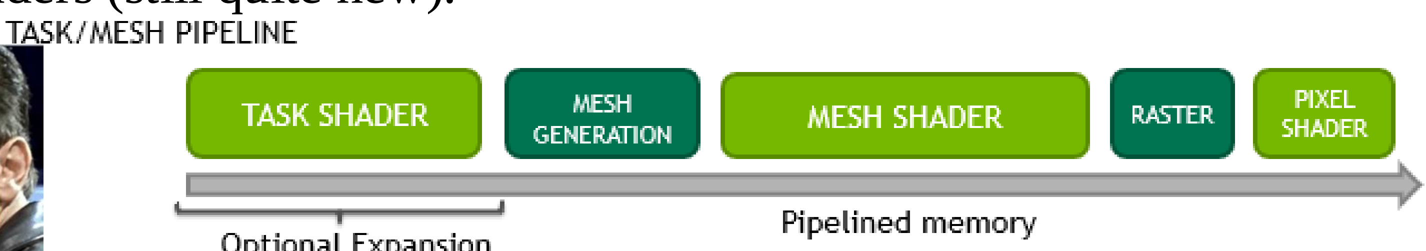
Full traditional pipeline:







Mesh shaders (still quite new):



<u>Compatibility</u>:
- OpenGL 4.5 extension
- DirectX 12 Ultimate
- Vulcan

Simple Application...

```
int main(int argc, char *argv[])
{
    // open window of size 512x512 with double buffering, RGB colors, and Z-buffering
    g_window = labhelper::init_window_SDL("OpenGL Lab 1”, 512, 512);
    initGL(); // Set up our shaderProgram and our vertexArrayObject
    while (true) {

        display(); // render our geometry

        SDL_GL_SwapWindow(g_window); // swap front/back buffer. Ie., displays the frame.


        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT || (event.type == SDL_KEYUP &&
                event.key.keysym.sym == SDLK_ESCAPE)) {
                    labhelper::shutDown(g_window);
                    return 0;
            }
        }
    }
    return 0;
}
```

```
void display(void)
{
    // The viewport determines how many pixels we are rasterizing to
    int w, h;
    SDL_GetWindowSize(g_window, &w, &h);
    glViewport(0, 0, w, h);   // Set viewport

    // Clear background
    glClearColor(0.2, 0.2, 0.8, 1.0);   // Set clear color  - for background
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clears the color buffer and the z-buffer

    glDisable(GL_CULL_FACE); // Both front and back face of triangles should be visible

    // DRAW OUR TRIANGLE(S)
    glUseProgram( shaderProgram );  // Shader Program. Sets what vertex/fragment shaders to use.
    // Bind the vertex array object that contains all the vertex data.
    glBindVertexArray(vertexArrayObject);
    // Submit triangles from currently bound vertex array object.
    glDrawArrays( GL_TRIANGLES, 0, 3 );        // Render 1 triangle (i.e., 3 vertices), starting at vertex 0.

    glUseProgram( 0 );  // "unsets" the current shader program. Not really necessary.
}
```

Lab 1 will teach you this, i.e., setting up a shader program and vertex arrays.

Example of a simple GfxObject class

```
class GfxObject {
public:
        load(“filename”); // Creates m_shaderProgram + m_vertexArrayObject
        render()
        {
                /* You may want to initiate more OpenGL states, e.g., for
                   textures (more on that in further lectures) */
                glUseProgram(m_shaderProgram);
                glBindVertexArray(m_vertexArrayObject);
                glDrawArrays( GL_TRIANGLES, 0, numVertices);
        };
private:
        uint      numVertices;
        Gluint    m_shaderProgram;
        GLuint    m_vertexArrayObject;
};
```

Example:

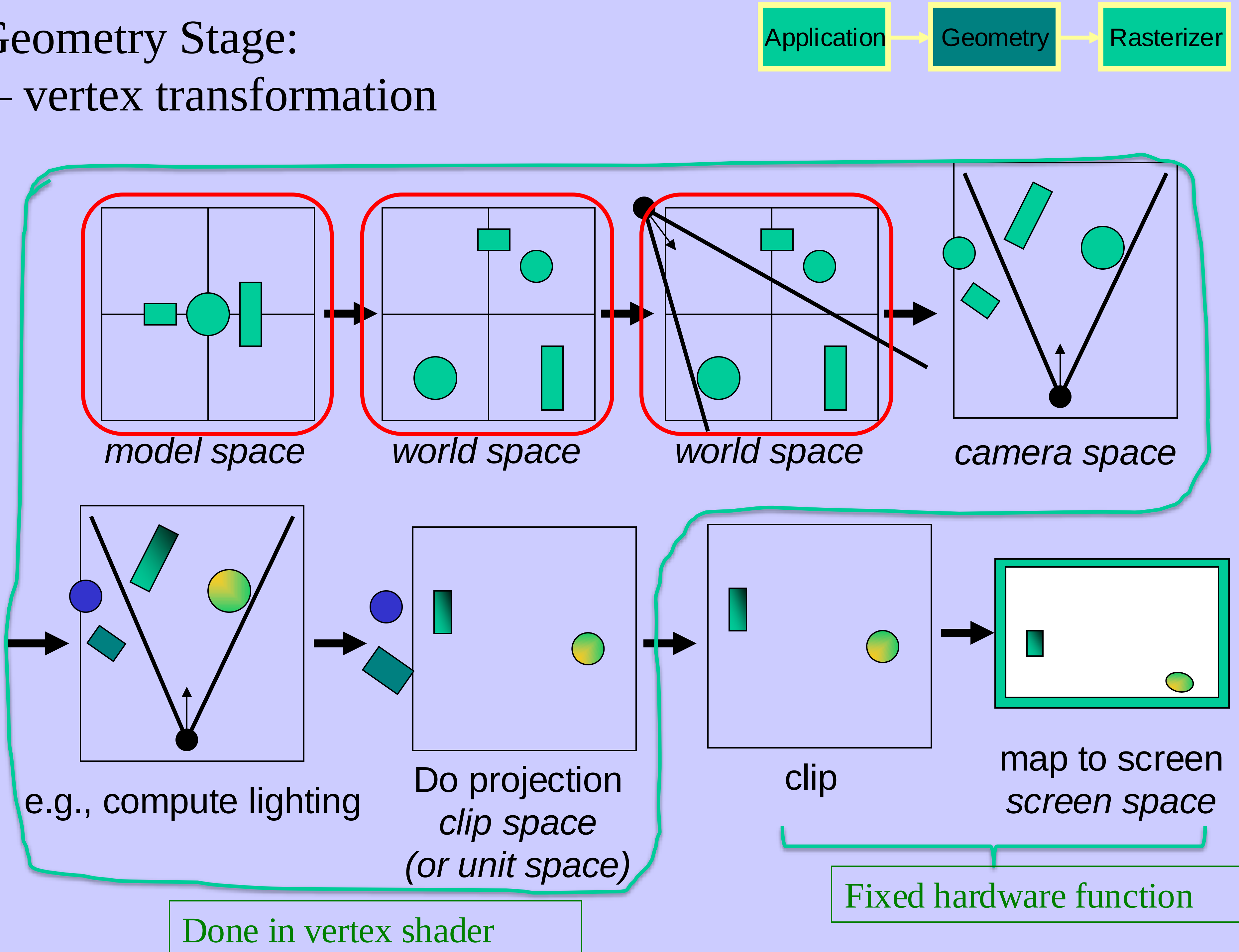
```
GfxObject myCoolObject;
myCoolObject.load(“filename”);


In display():
        myCoolObject.render();
```

The Geometry stage and Rasterizer stage in more detail

Rewind!
Let’s take a closer look

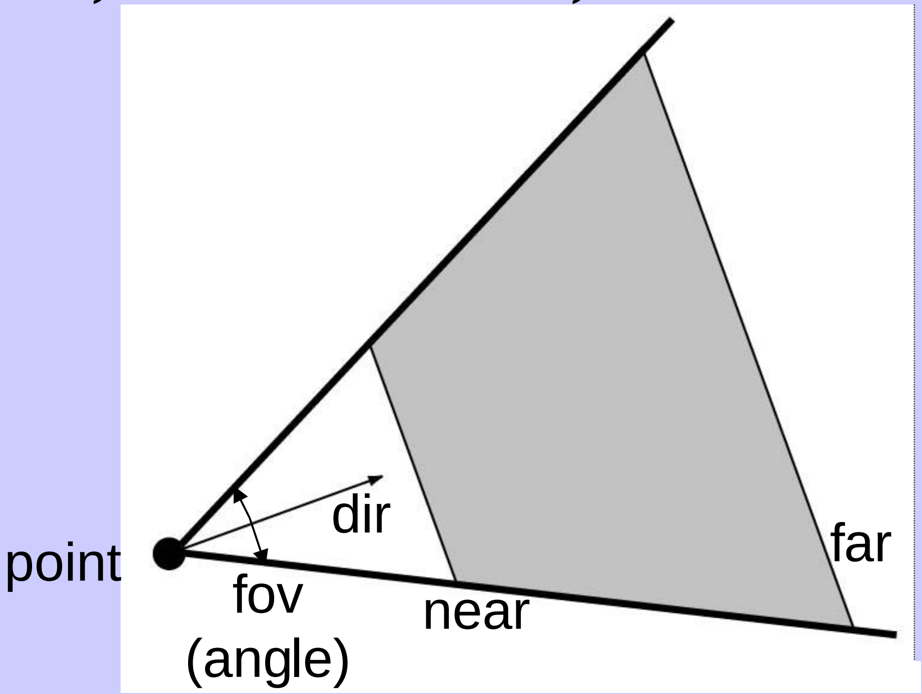
- The programmer ”sends” down primtives to be rendered through the pipeline (using API calls)
- The geometry stage does per-vertex operations
- The rasterizer stage does per-pixel operations
- Next, scrutinize geometry and rasterizer

Geometry Stage:
– vertex transformation



Virtual Camera

- Defined by position, direction vector, up vector, field of view, near and far plane.

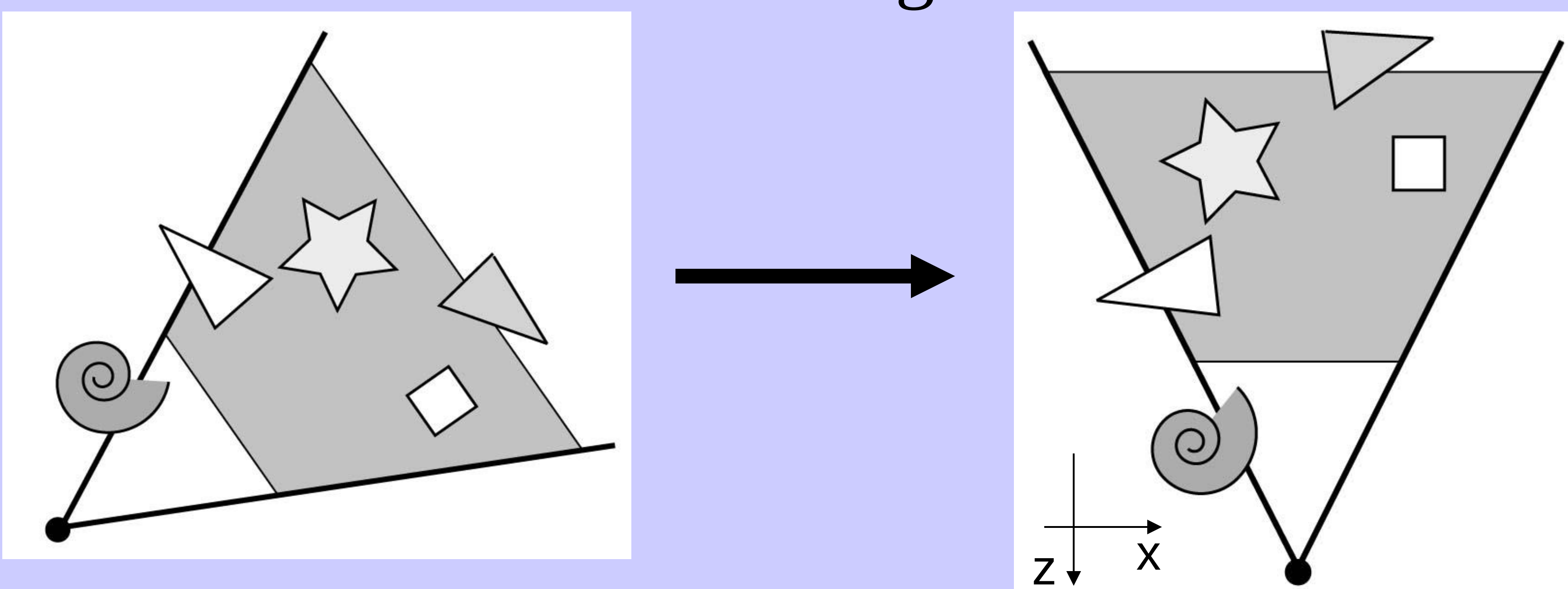


- Create image of geometry inside gray region
- Used by OpenGL, DirectX, ray tracing, etc.

Geometry Stage:
– vertex transformation

The view transform

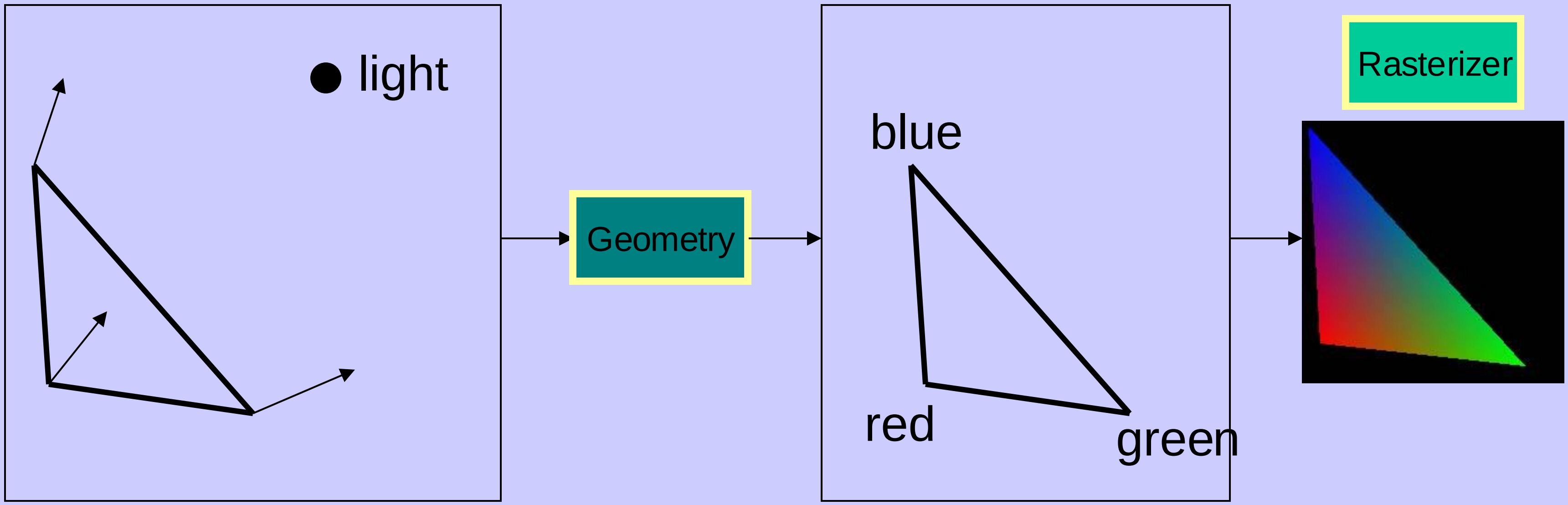
- You can move the camera in the same manner as objects
- But apply inverse transform to objects, so that camera looks down negative z-axis



Geometry Stage:
– vertex transformation

Lighting

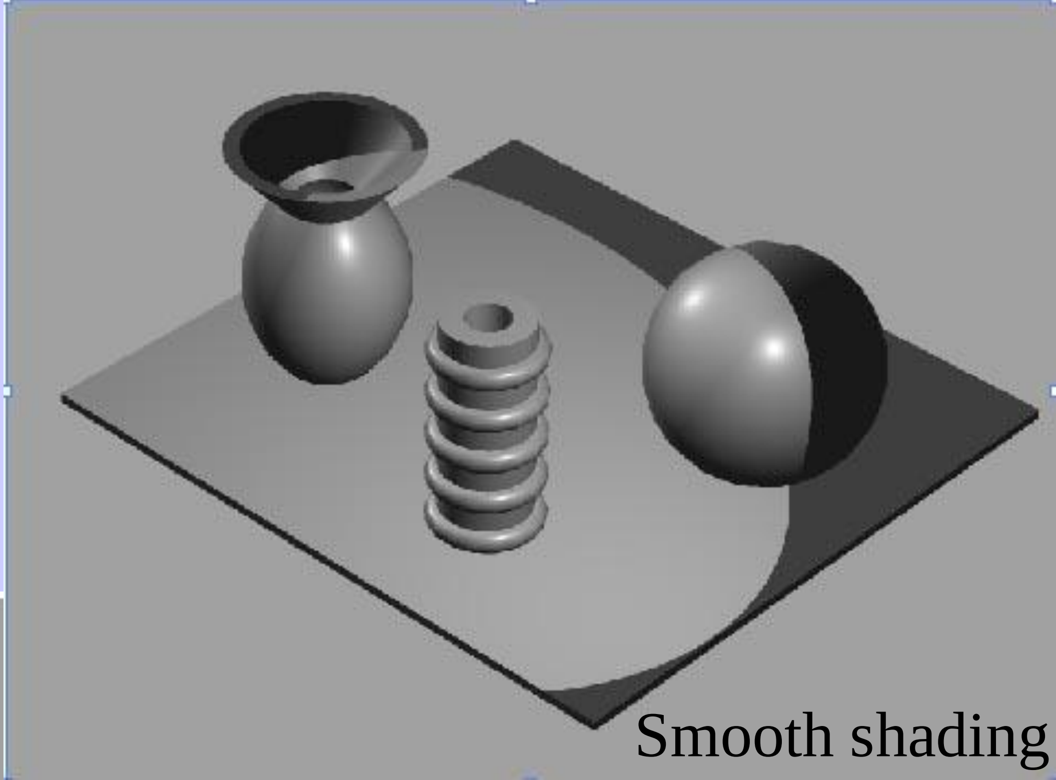
- Compute full or partial lighting information for fragment shader (e.g., light direction, normal, and vertex position)

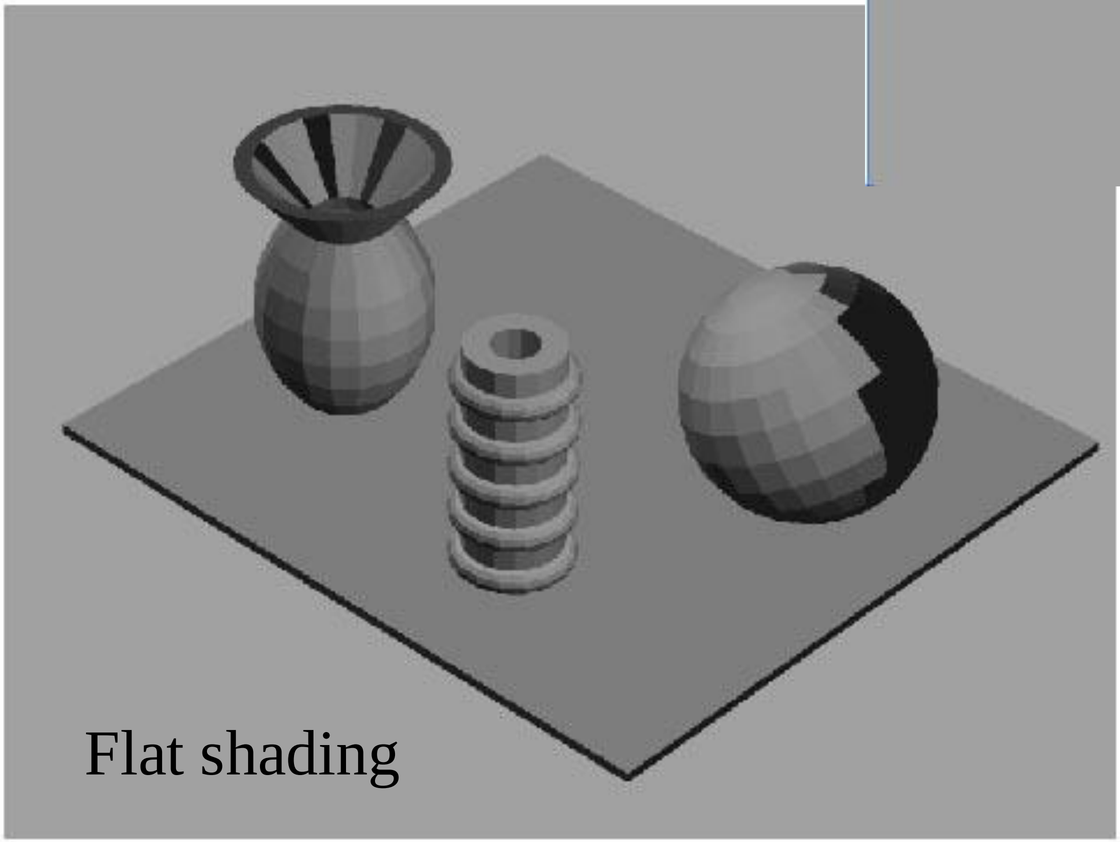


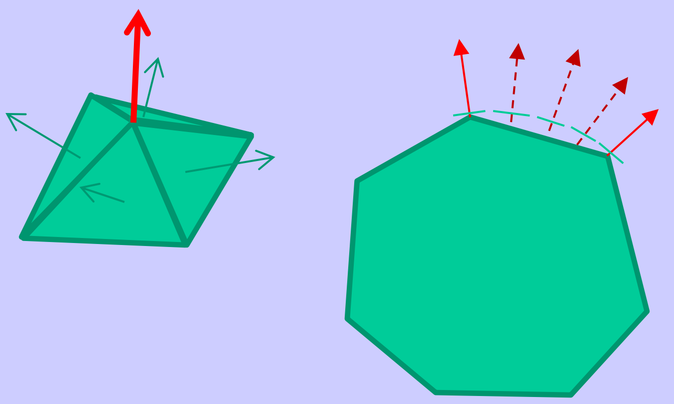
- Much more about this in later lecture

Why a normal per vertex?

to shade as a curved surface
although triangle is flat



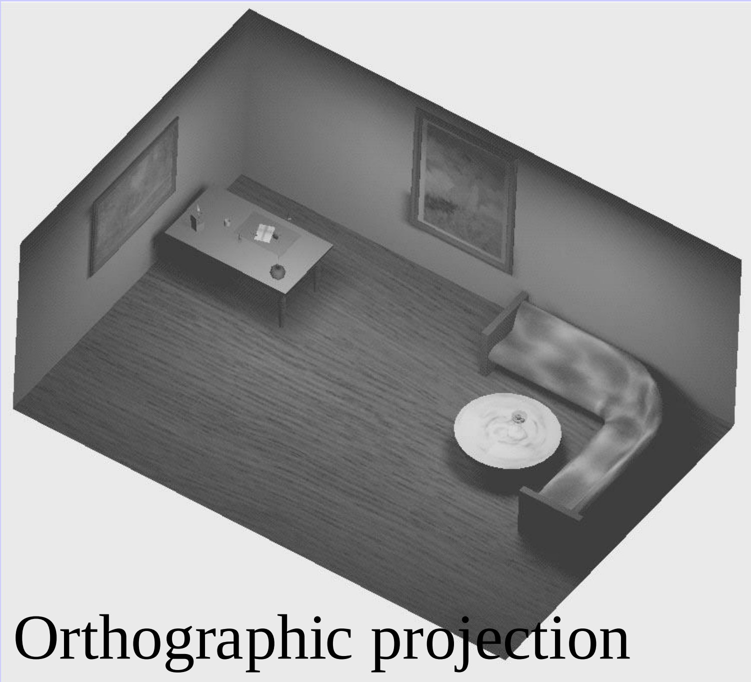




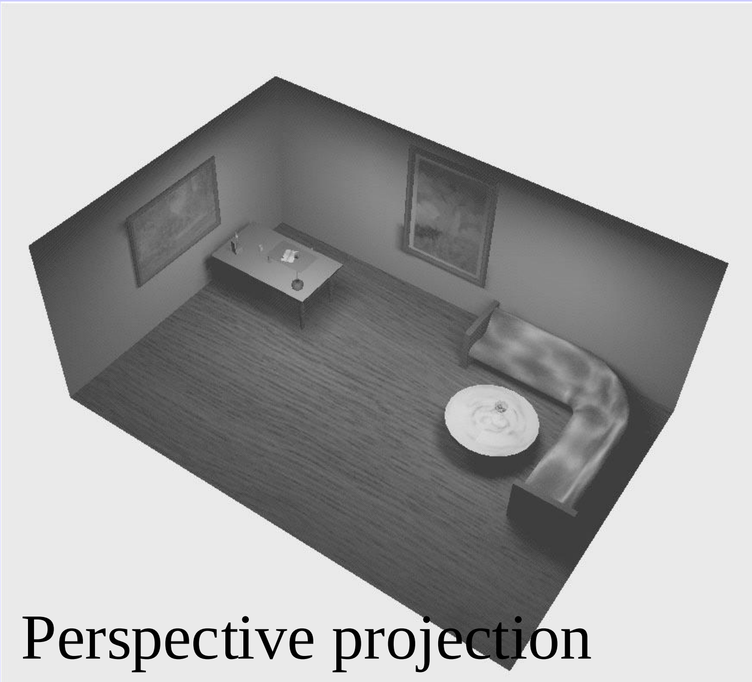
Geometry Stage:
– vertex transformation

Projection

- Two major ways to do it
  - Orthographic (useful in few applications)
  - Perspective (most often used)
    - Mimics how humans perceive the world, i.e., objects' apparent size decreases with distance



Orthographic projection

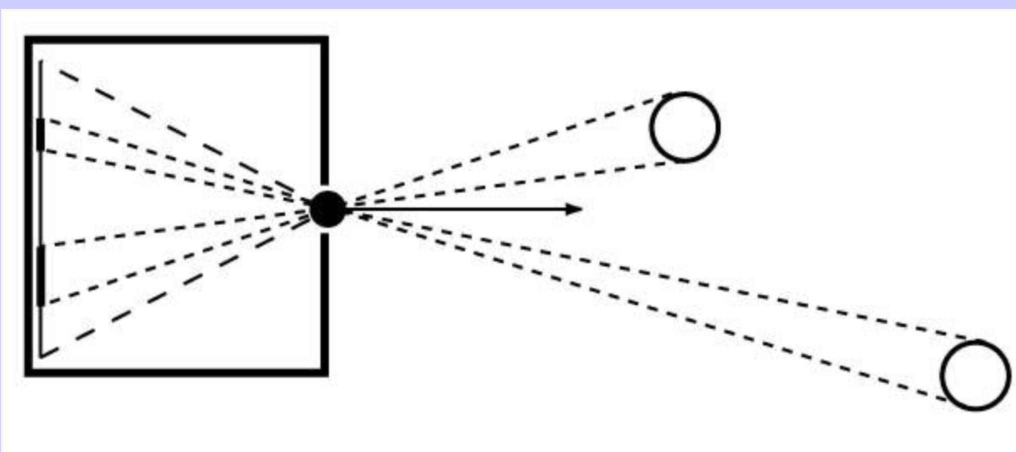


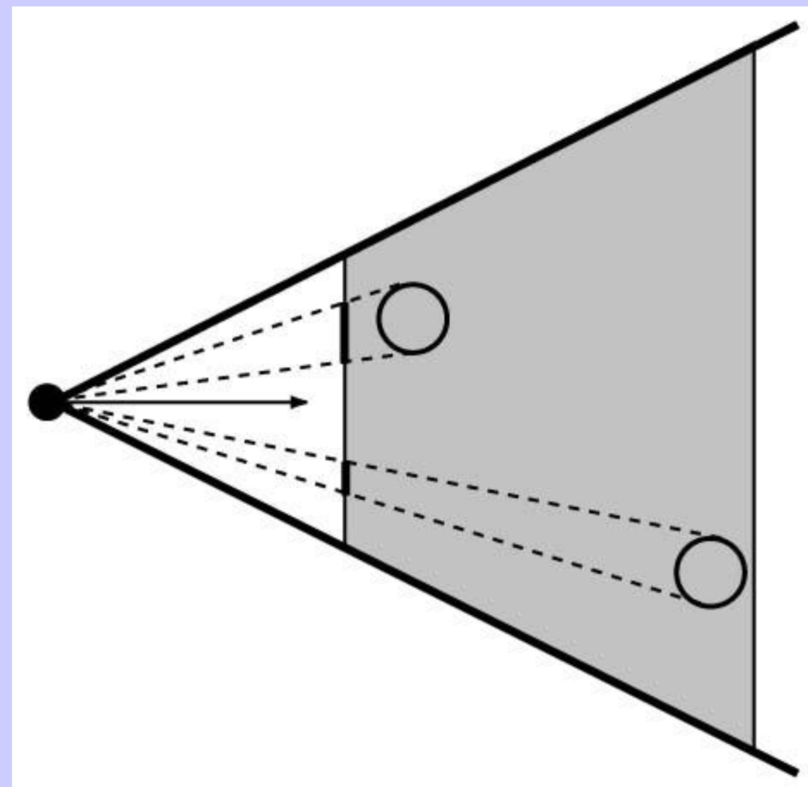
Perspective projection

Geometry Stage:
– vertex transformation

Projection

- Also done with a matrix multiplication!
- Pinhole camera (left), analog used in CG (right)

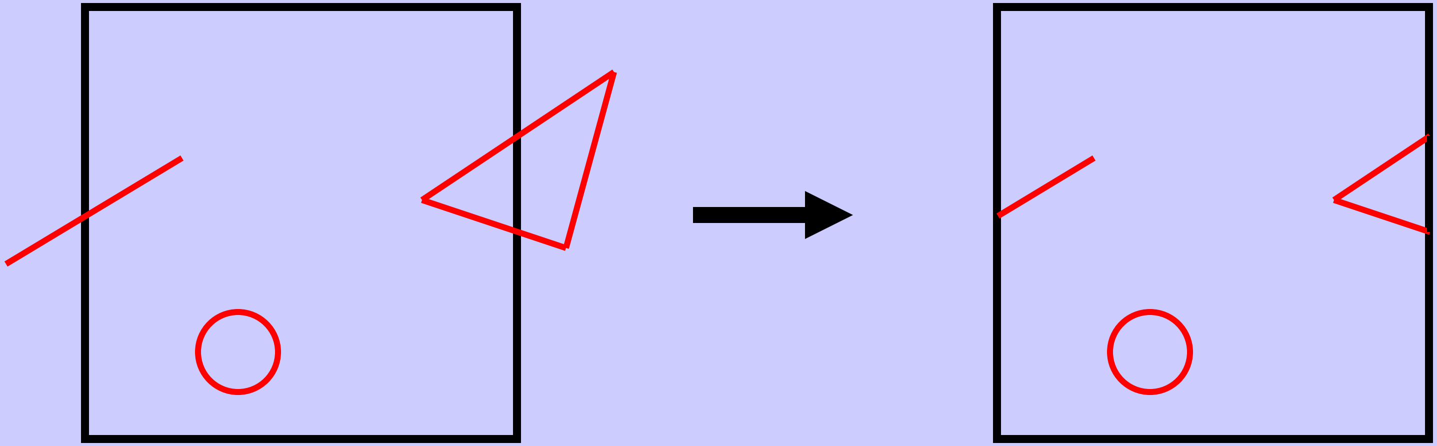




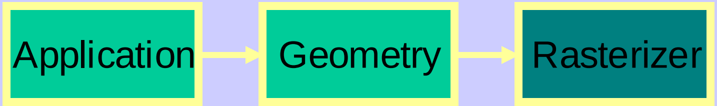
GEOMETRY

Clipping and Screen Mapping

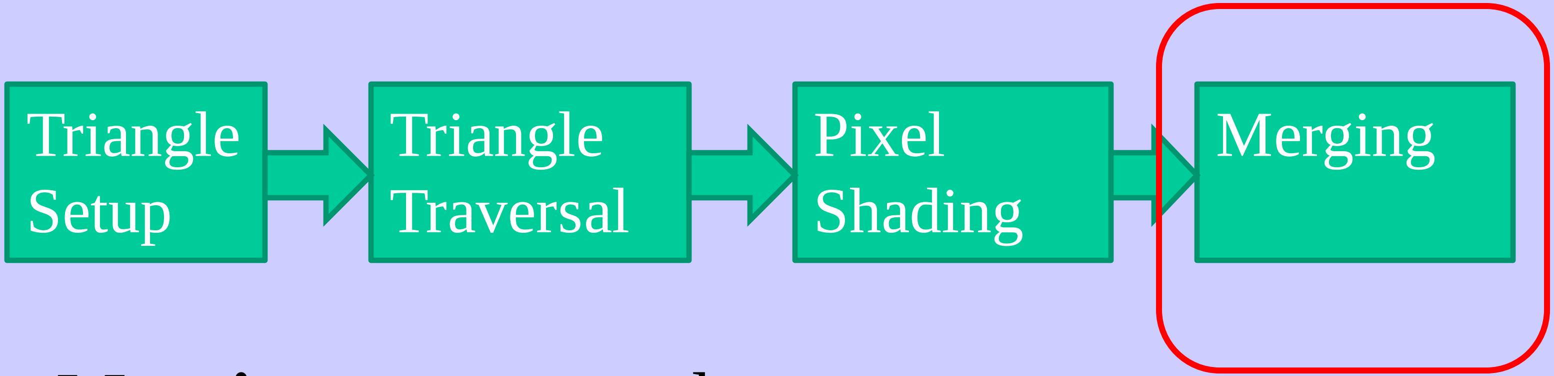
- Square (cube) after projection
- Clip primitives to square



- Screen mapping, scales and translates the square so that it ends up in a rendering window
- These ”screen space coordinates” together with Z (depth) are sent to the rasterizer stage



The rasterizer stage



Merging - output color to screen: includes for instance…

- **Z-buffering**
  - Do not overdraw a pixel with content that is further from the camera than the pixel's current content
- **Doublebuffering**
  - Use a front buffer that is displayed and a backbuffer that we still draw to.

The default frame buffer:

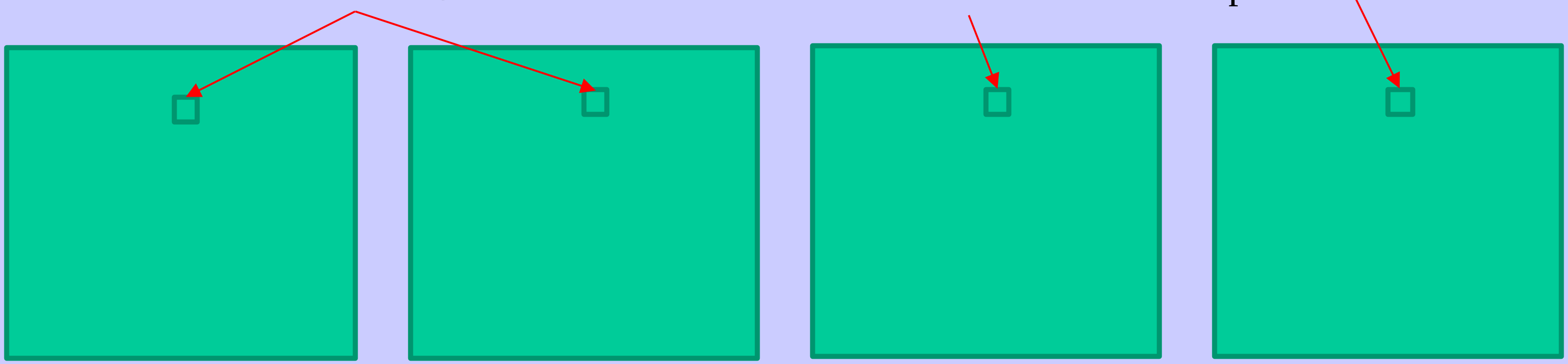
Typically: Front + Back **color** buffers + Z buffer + (Stencil buffer)

These are memory buffers, e.g., in GPU RAM.

Stores rgb(a) value per pixel.
Default: 8 bits per r,g,b channel.

Stores fragment's depth value per pixel, typically: (16), 24, or 32 bits.

Stencil buffer can be asked for. 8-bits per pixel.



Front Color buffer (rgb buffer)

Is the most recent fully finished drawn frame.
Is displayed.

Back Color buffer (rgb buffer)

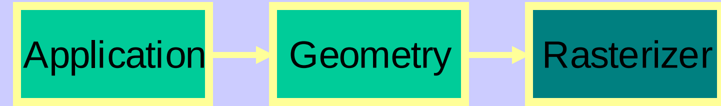
Is the color buffer we still draw to.
Not displayed yet.

Z buffer (depth)

To resolve visibility between triangles

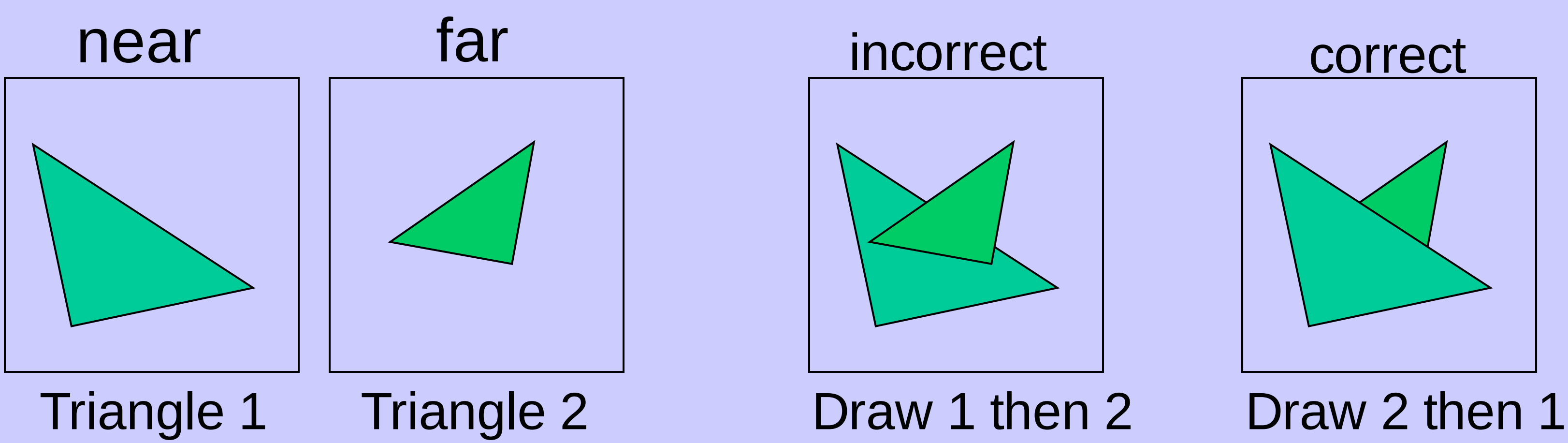
Stencil buffer (8-bits)

Used for masking rendering to only where pixel's stencil value = some specific value.



The rasterizer stage

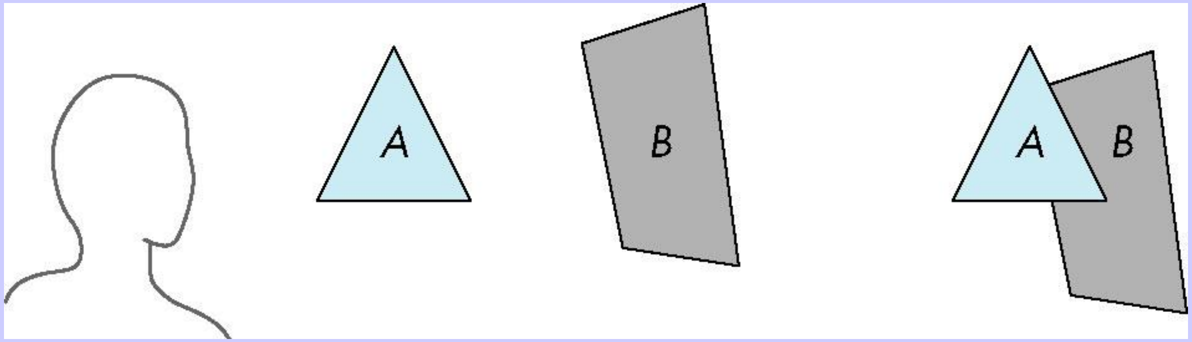
- A triangle that is covered by a more closely located triangle should not be visible
- Assume two equally large tris at different depths



Old way before z-buffers:

Painter’s Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over



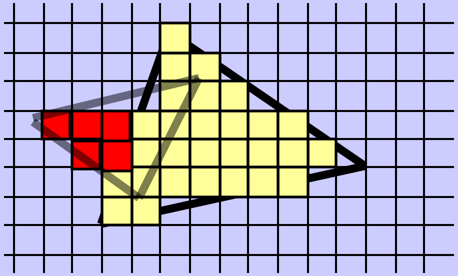
B behind A as seen by viewer

Fill B then A

- Requires ordering of polygons first
  - –O(n log n) calculation for ordering
  - –Not every polygon is either in front or behind all other polygons

I.e., : Sort all triangles and render them back-to-front.

The rasterizer stage



Z-buffering:

- Would be nice to avoid sorting…
- The Z-buffer (aka depth buffer) solves this
- Can render in any order

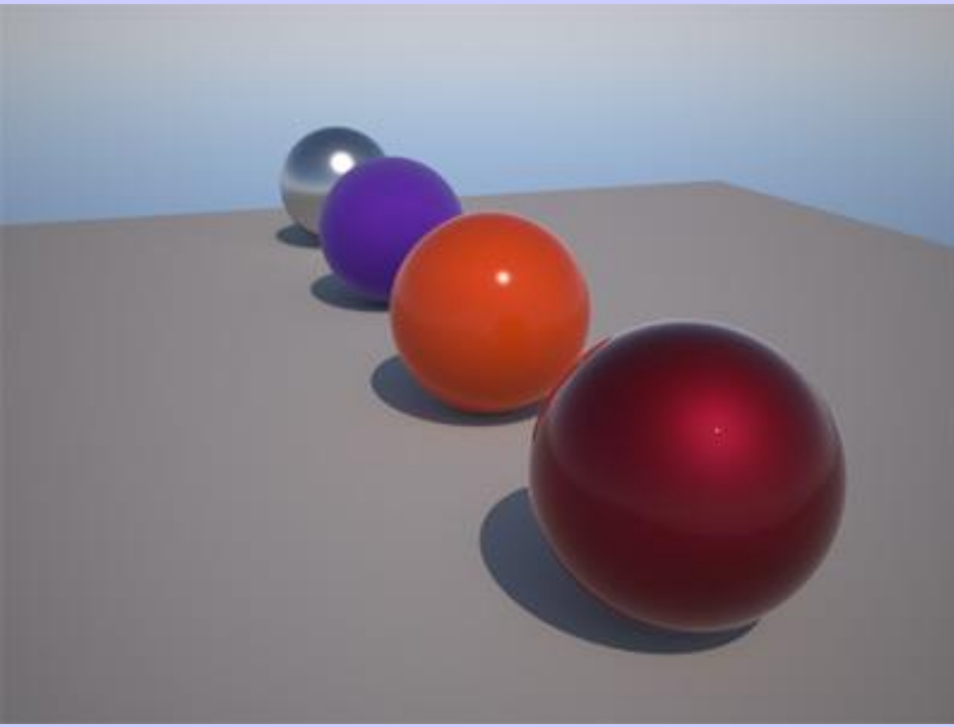
Idea - storing closest fragment-z (triangle depth) at each pixel:

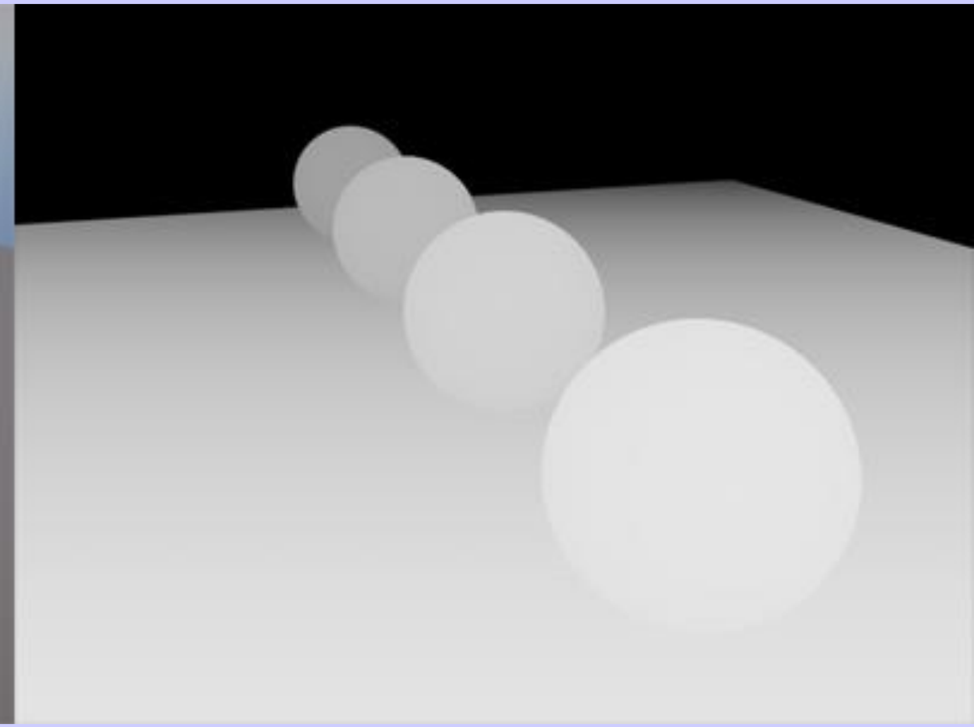
- When rasterizing, compute the fragment's z-value.
- Compare this to pixel's Z-buffer value
- If fragment is closer, then replace color-buffer's and Z-buffer's value
- Else do nothing (discard the fragment)

- Z-buffer stores, for each pixel, the closest fragment's z-value.
- Color buffer stores, for each pixel, the color value.

I.e., do not overdraw a pixel with content that is further from the camera than its current content

Z-buffer





The color buffer

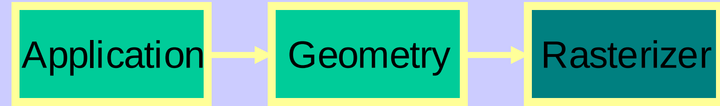
The z-buffer
(= depth buffer)

Z-buffer

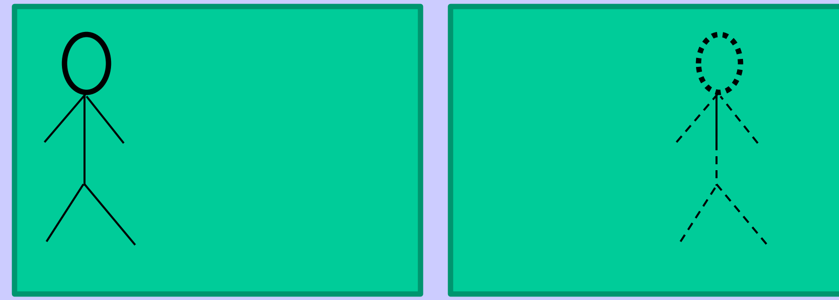
Z-Buffer Algorithm

- Use a buffer called the z or depth buffer to store the depth of the closest fragment at each pixel rasterized so far
- If a new fragment's depth is closer to camera than pixel's z-buffer value,
  - replace pixel's color and z-buffer value.

The rasterizer stage



Double buffering:



Front buffer
(rgb color buffer)

Latest fully finished
drawn frame.

Back buffer
(rgb color buffer)

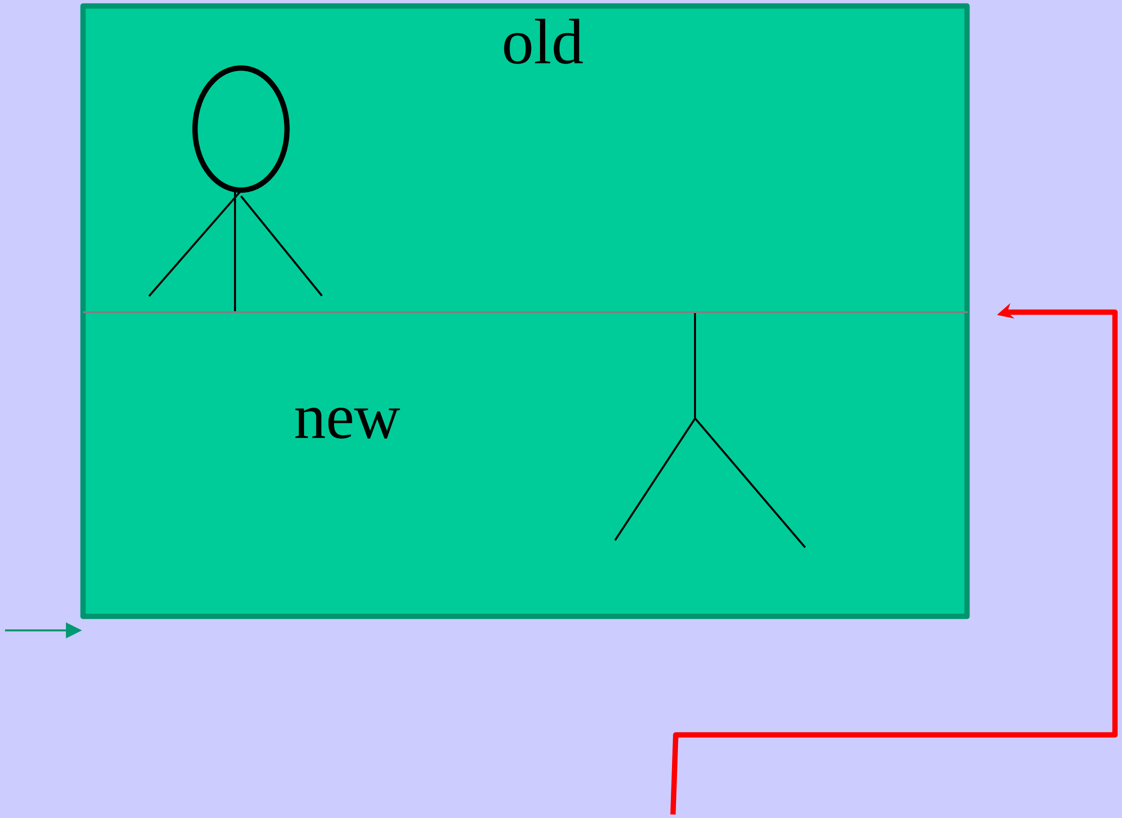
Color buffer we draw to.
Not displayed yet.

- We do not want to show the image until its drawing is finished.
- The front buffer is displayed
- The back buffer is rendered to
- When new image has been created in back buffer, swap the Front-/Back-buffer pointers.
- Use vsync. Else, screen tearing will occur…
  i.e., when the swap happens in the middle of the screen with respect to the screen refresh rate.

The rasterizer stage

Double buffering – *screen tearing:*

Monitors update the screen line by line from top to bottom, and each line from left to right.



Use vsync to swap here:

Front- and back-buffer pointers swapped “within the monitor’s update” of the screen.

Example if the swap happens here (w.r.t the screen refresh rate). Solution: use vsync to swap buffers after monitor has updated the full screen. See page 1011-1012.

Screen Tearing

Swapping
back/front buffers

vblank

Screen tearing is solved by using V-Sync.

V-Sync: swap front/back buffers during vertical blank (vblank) instead.

Screen Tearing

- Despite the gorgeous graphics seen in many of today's games, there are still some highly distracting artifacts that appear in gameplay despite our best efforts to suppress them. The most jarring of these is screen tearing. Tearing is easily observed when the mouse is panned from side to side. The result is that the screen appears to be torn between multiple frames with an intense flickering effect. Tearing tends to be aggravated when the framerate is high since a large number of frames are in flight at a given time, causing multiple bands of tearing.
- **Vertical sync (V-Sync) is the traditional remedy to this problem**, but as many gamers know, V-Sync isn't without its problems. The main problem with V-Sync is that when the framerate drops below the monitor's refresh rate (typically 60 fps), the framerate drops disproportionately. For example, dropping slightly below 60 fps results in the framerate dropping to 30 fps. This happens because the monitor refreshes at fixed internals (although an LCD doesn't have this limitation, the GPU must treat it as a CRT to maintain backward compatibility) and V-Sync forces the GPU to wait for the next refresh before updating the screen with a new image. This results in notable stuttering when the framerate dips below 60, even if just momentarily.

A Swap Chain

Tripple buffering - or even more intermediate buffers

So, GPU does not have to wait until the swap for starting rendering the next frame.

What is important:

- Understand the Application-, Geometry- and Rasterization Stage
- Correlation to hardware
- Z-buffering, double buffering, screen tearing

BONUS

Simple Application...

OLD WAY

OpenGL 1.1

```
#ifdef WIN32
#include <windows.h>
#endif

#include <GL/glut.h>                    // This also includes gl.h

static void drawScene(void)
{
    glColor3f(1,1,1);


    glBegin(GL_POLYGON);
        glVertex3f( 4.0, 0, 4.0);
        glVertex3f( 4.0, 0,-4.0);
        glVertex3f(-4.0, 0,-4.0);
    glEnd();
}
```

Usually this and next 2 slides are put in the same file main.cpp

Simple Application

BONUS
Old way
OpenGL 1.1

```
void display(void)
{
    glClearColor(0.2, 0.2, 0.8, 1.0);           // Set clear color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clears the color buffer
                                                           and the z-buffer
    int w = glutGet((GLenum)GLUT_WINDOW_WIDTH);
    int h = glutGet((GLenum)GLUT_WINDOW_HEIGHT);
    glViewport(0, 0, w, h);                     // Set viewport

    glMatrixMode(GL_PROJECTION);                // Set projection matrix
    glLoadIdentity();
    gluPerspective(45.0,w/h, 0.2, 10000.0);     // FOV, aspect ratio, near, far

    glMatrixMode(GL_MODELVIEW);                 // Set modelview matrix
    glLoadIdentity();

    gluLookAt(10, 10, 10,                       // look from
              0, 0, 0,                          // look at
              0, 0, 1);                         // up vector

    drawScene();
    glutSwapBuffers();  // swap front and back buffer. This frame will now been displayed.
}
```

Changing Color per Vertex

BONUS
Old way
OpenGL 1.1

```
static void drawScene(void)
{
    // glColor3f(1,1,1);
    glBegin(GL_POLYGON);
        glColor3f(1,0,0);
        glVertex3f( 4.0, 0, 4.0);

        glColor3f(0,1,0);
        glVertex3f( 4.0, 0,-4.0);

        glColor3f(0,0,1);
        glVertex3f(-4.0, 0,-4.0);
    glEnd();
}
```