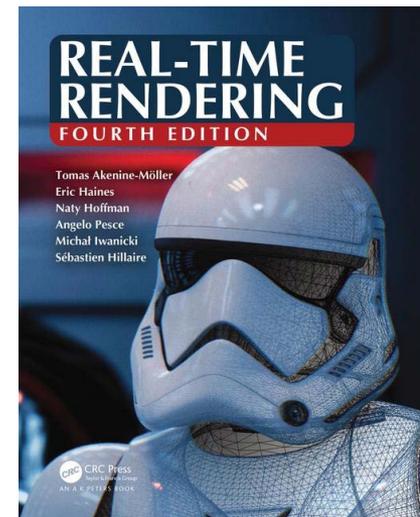


TDA362/DIT224 – Computer Graphics



Teacher: Ulf Assarsson
Chalmers University of Technology



This Course

- Algorithms!



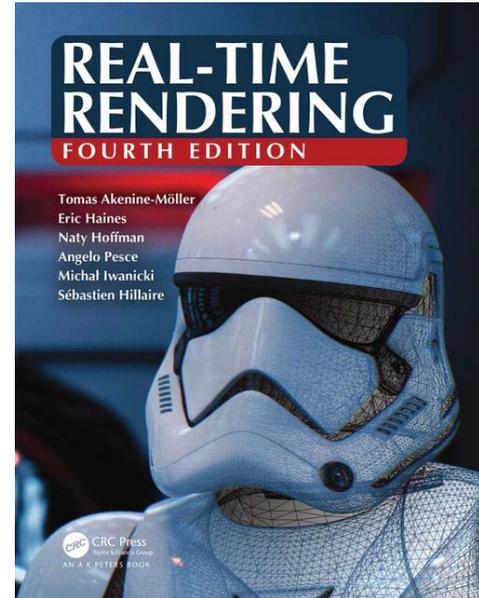
Understanding of
Ray Tracing

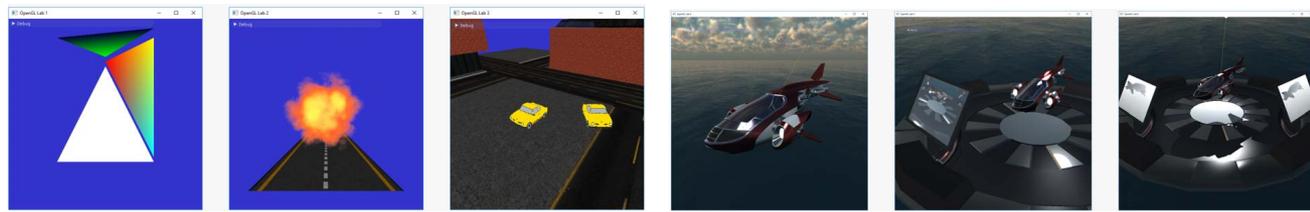


Real-time Rendering

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”

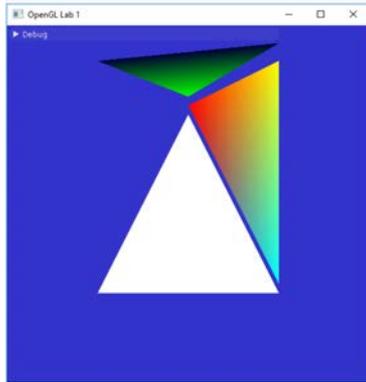




Tutorials

- 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



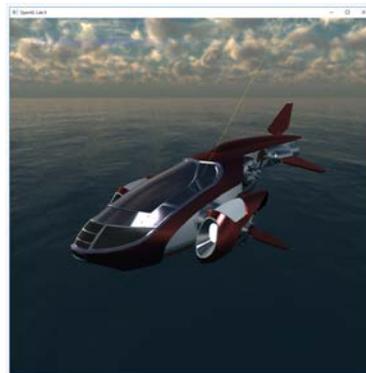
Rendering a triangle



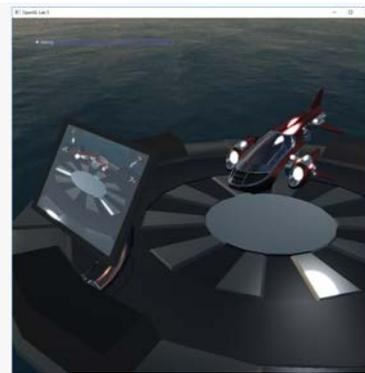
Textures



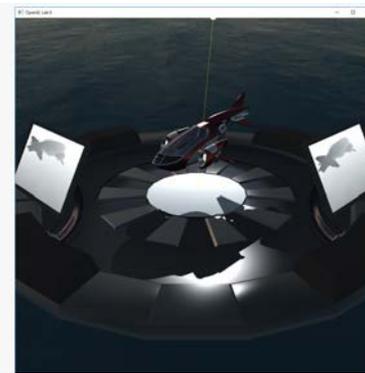
Animation



Shading



Render to textures



Shadow maps

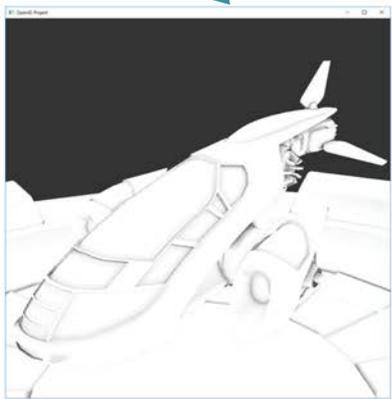
Project

Choose at least 1 from:

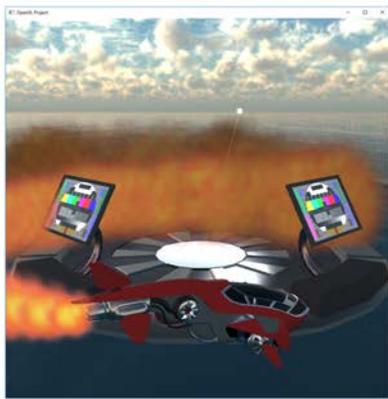
Project

Real-time rendering

Offline rendering



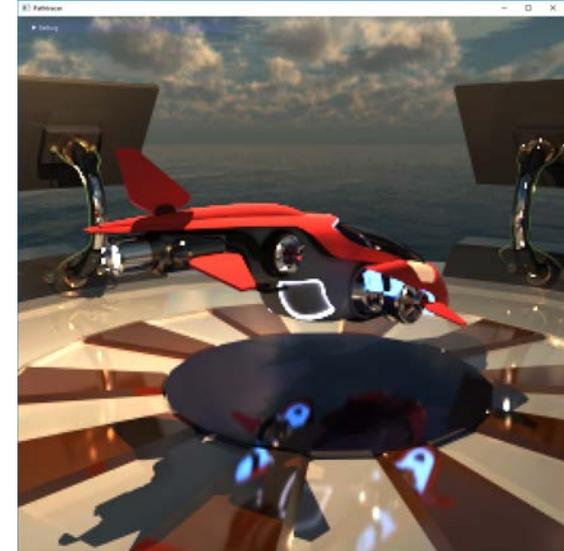
or



or



or



Path Tracing

Screen-space
ambient occlusion

Particle System

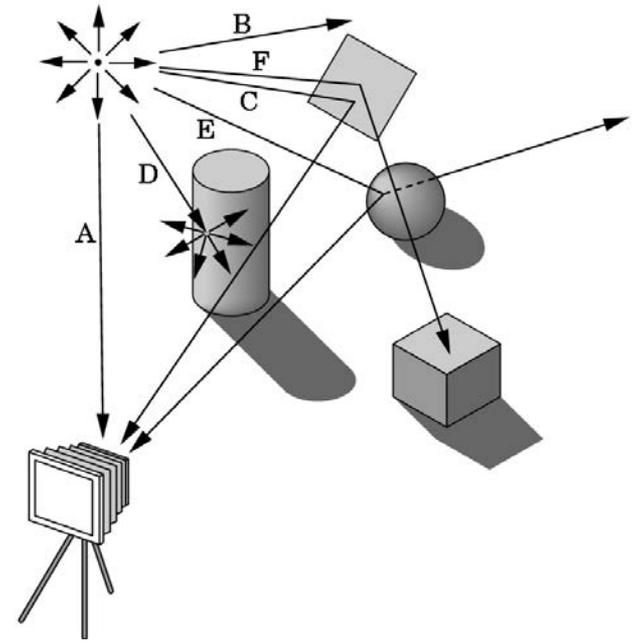
Height field

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 + 5: Thursday week 3.
 - Lab 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.

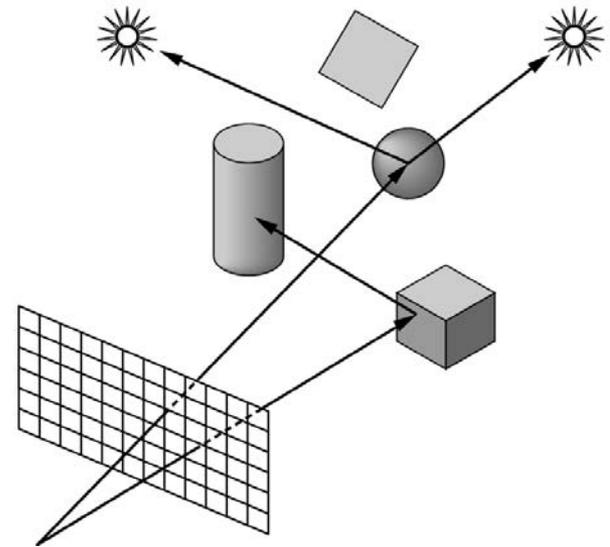
Tracing Photons

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



Other Physical Approaches

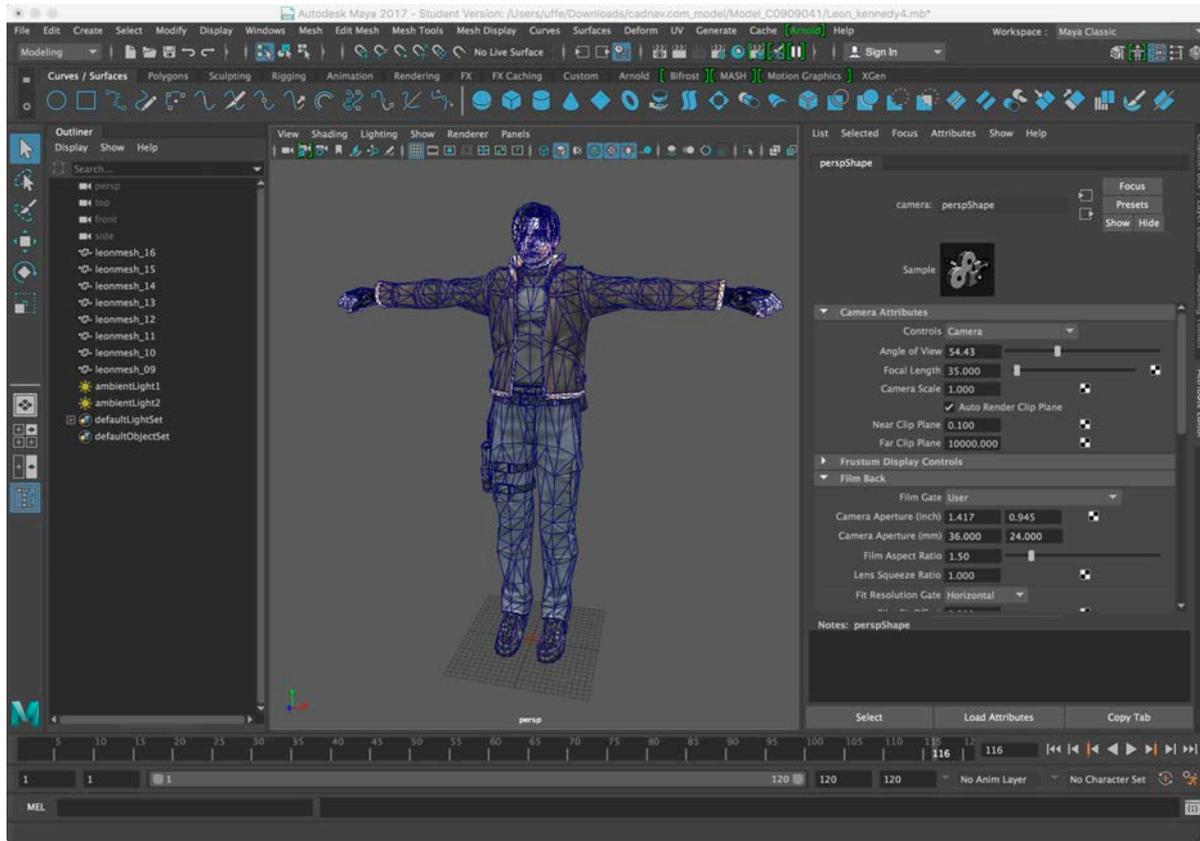
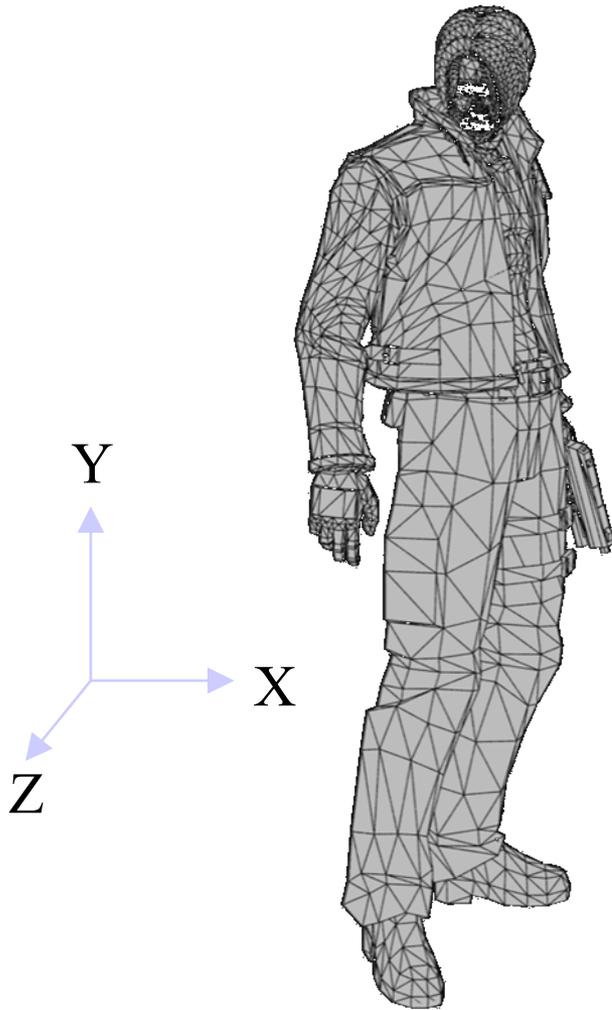
- **Ray tracing:** follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
 - Can handle global effects
 - Multiple reflections
 - Translucent objects
 - Faster but still rather slow



Real-Time Rendering

Overview of the Graphics Rendering Pipeline and OpenGL

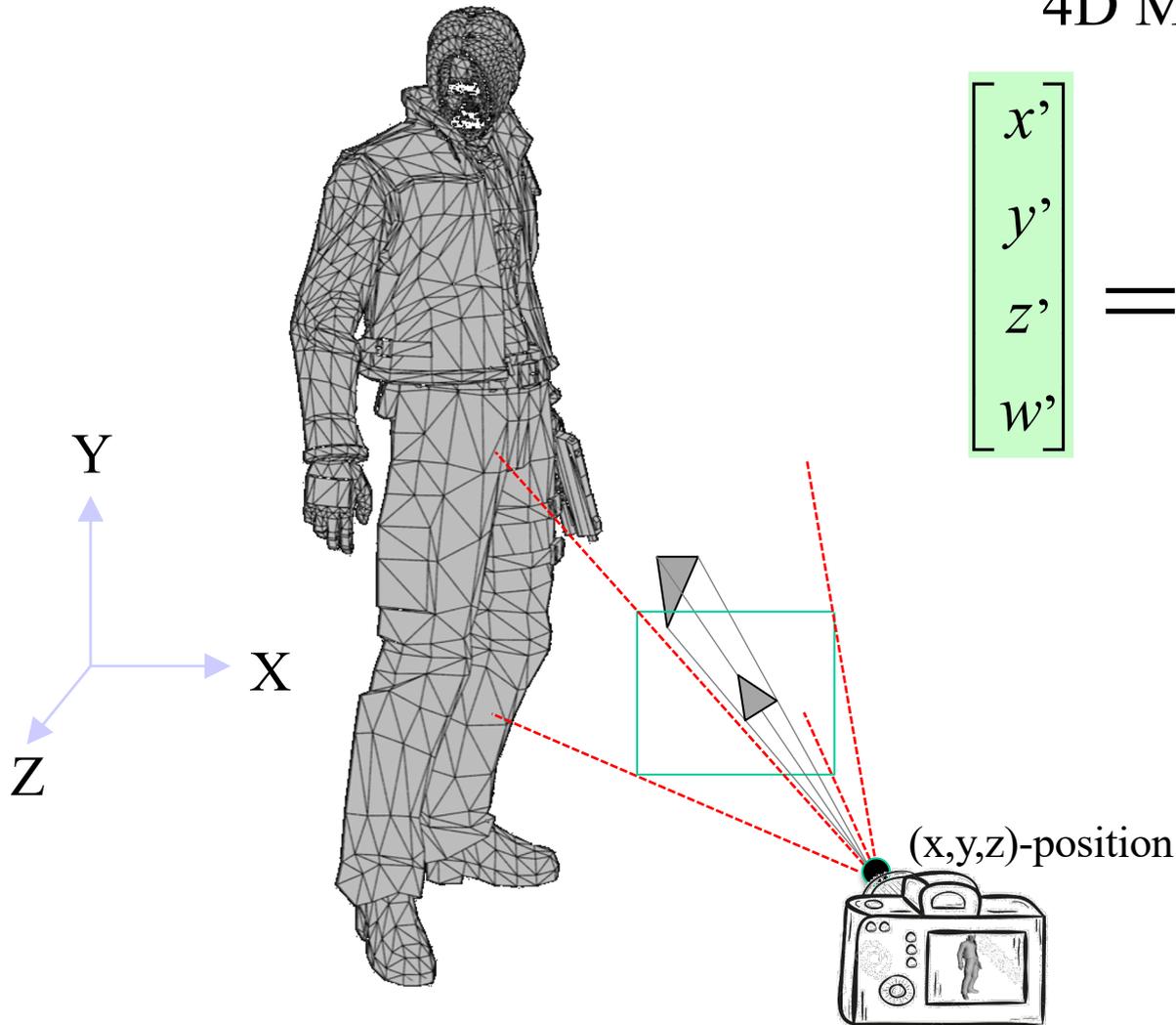
3D-models: surfaces are constructed by triangles.



4926 triangles

Why triangles?

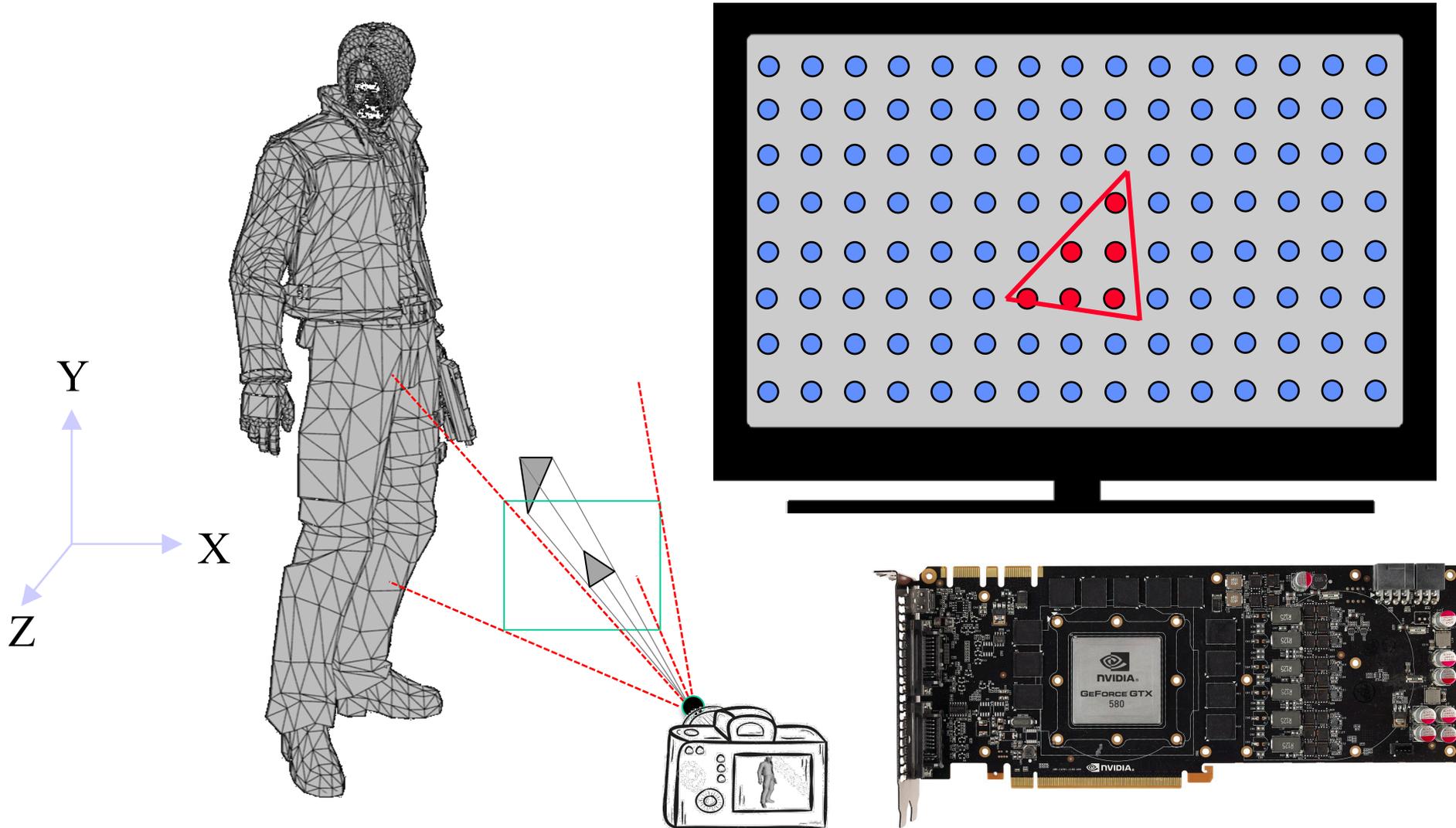
Each triangle is projected onto the image plane using a virtual camera.



4D Matrix Multiplication

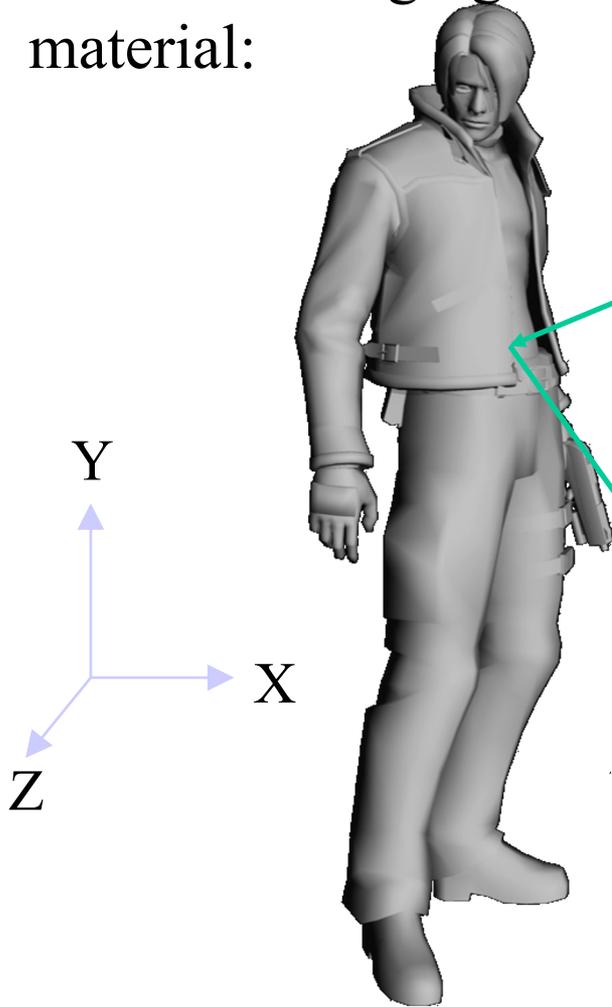
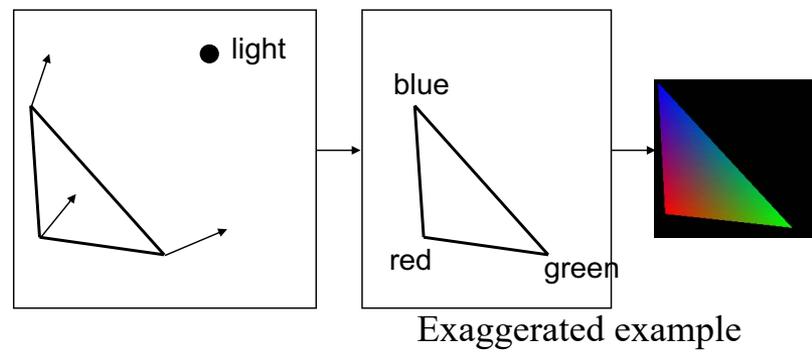
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} s_x & \bullet & \bullet & t_x \\ \bullet & s_y & \bullet & t_y \\ \bullet & \bullet & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

The graphics card draws the triangles onto the screen.



How compute pixel color?

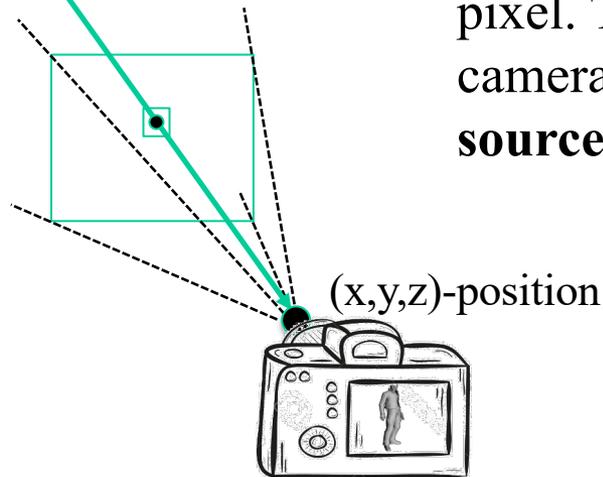
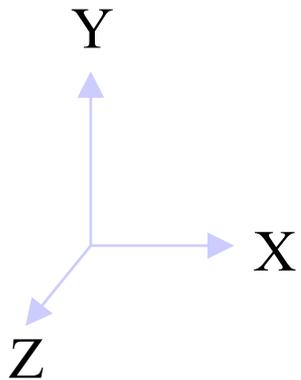
Use some *shading* model based on surrounding light and triangle's material:



 (x,y,z) Light source

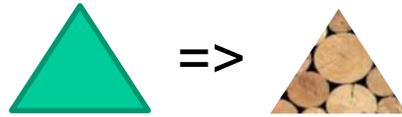
At rendering (for each frame):

- The graphics card computes, per pixel (or per vertex and using interpolation per pixel), the incoming light to the camera through each pixel. This **incoming light** to a camera pixel **depends on the light sources and the triangle's material.**

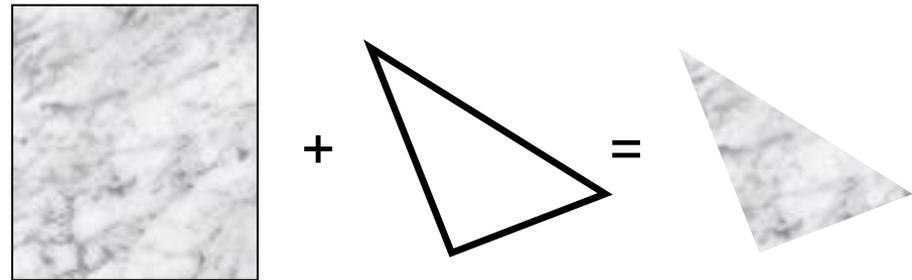


Triangle colors:

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.

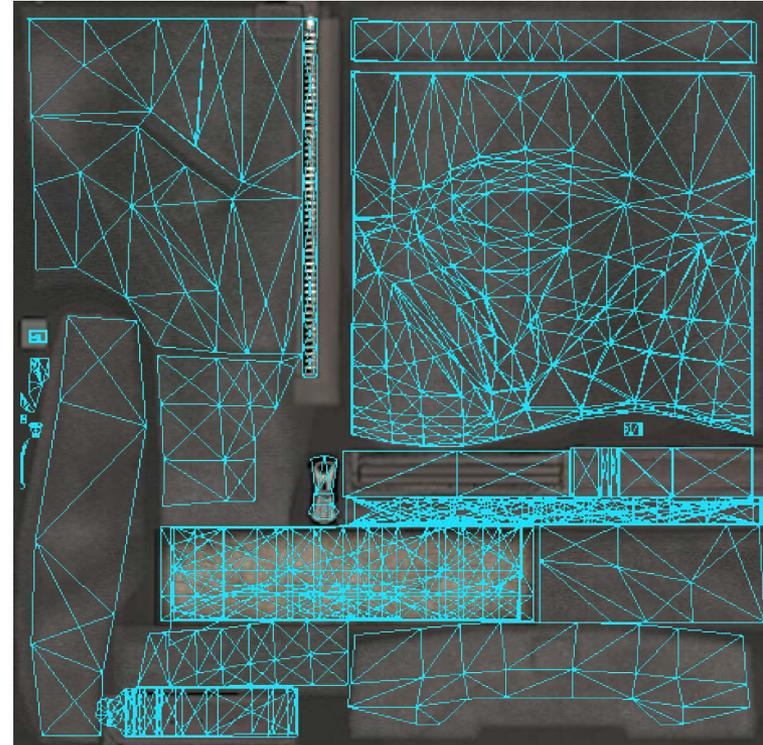
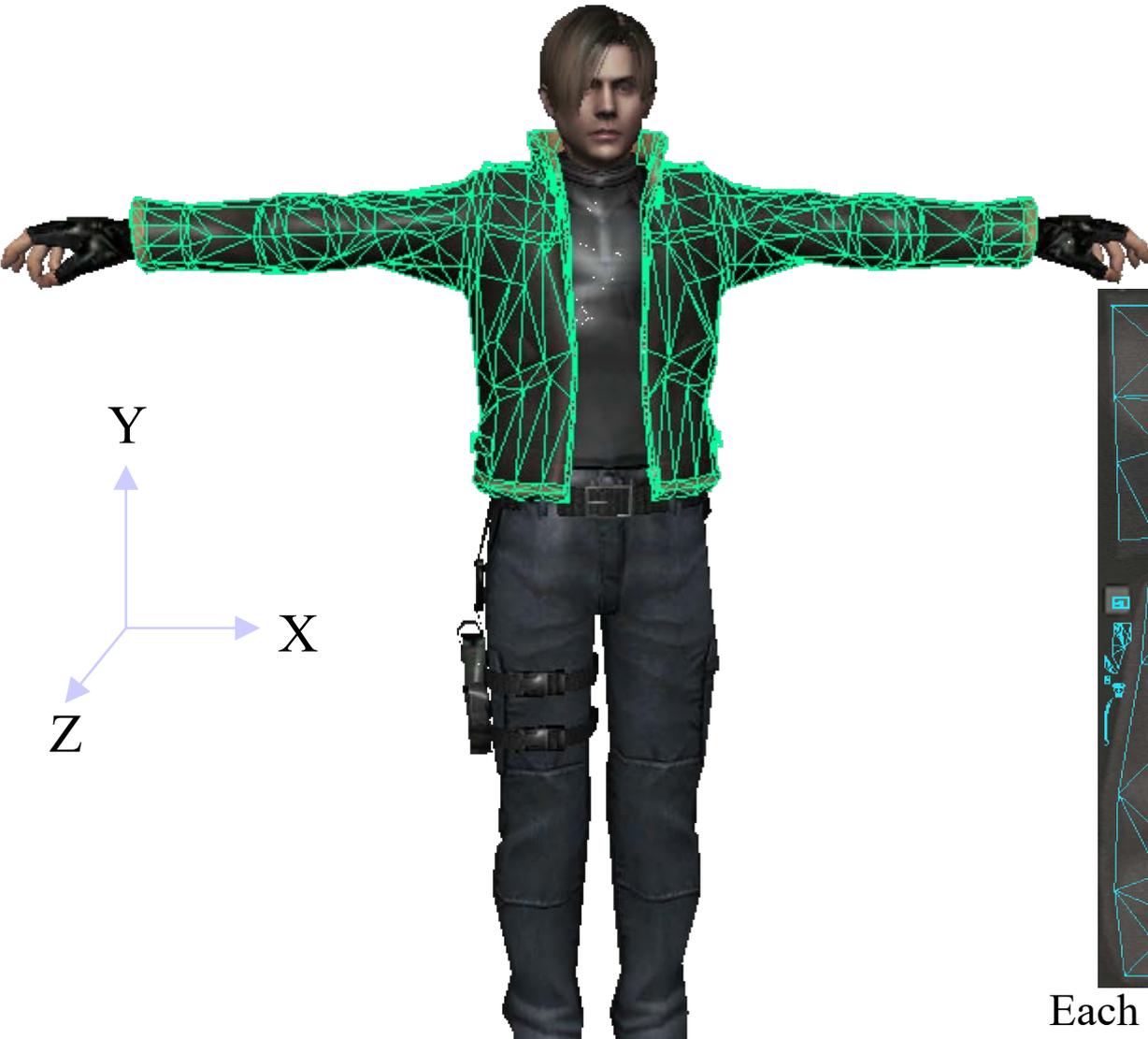


- The **texture** color is modulated (often just multiplied) with the light intensity to get the final pixel color.



Specify which part of the texture that each triangle covers.

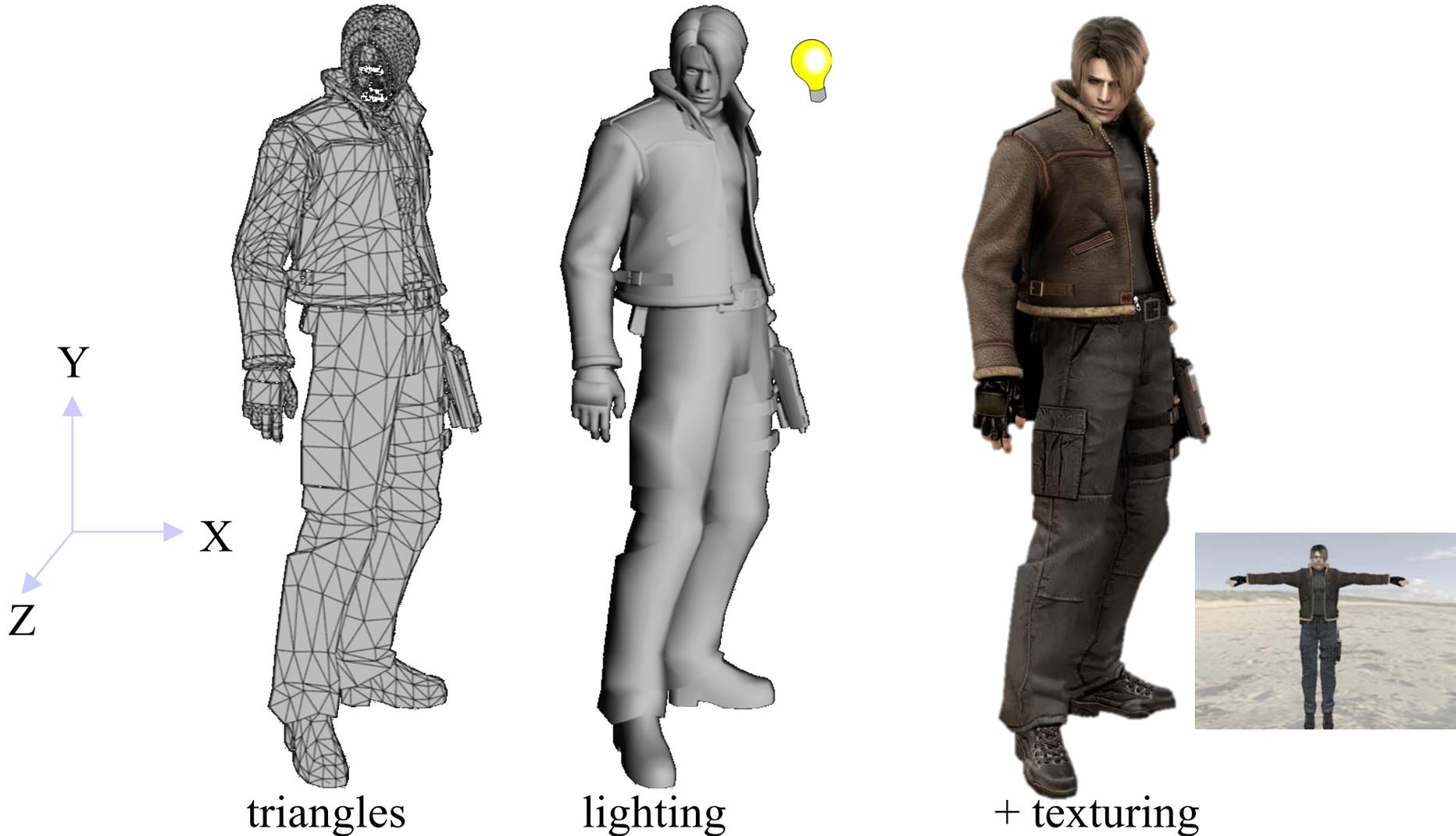
Texture Maps



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.

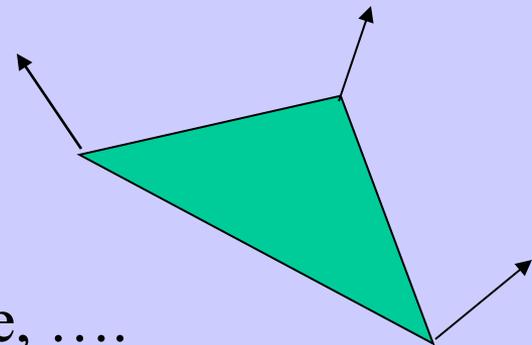


The Graphics Rendering Pipeline

The Application stage, geometry stage, and rasterizer stage

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

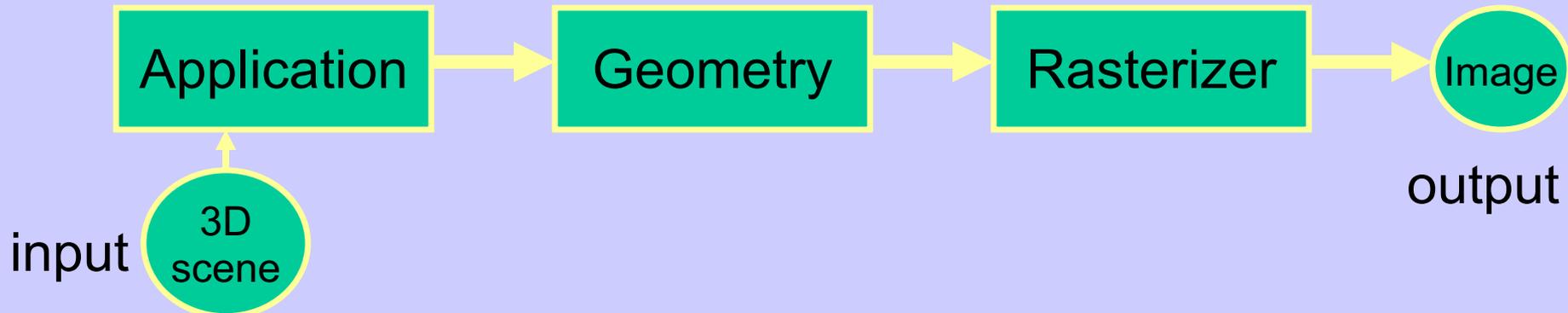
- First, of all to take a picture, it takes a camera – a virtual one.
 - Decides what should end up in the final image
- A 3D scene is:
 - Geometry (triangles, lines, points, and more)
 - Light sources
 - Material properties of geometry
 - Colors, shader code ,
 - Textures (images to glue onto the geometry)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

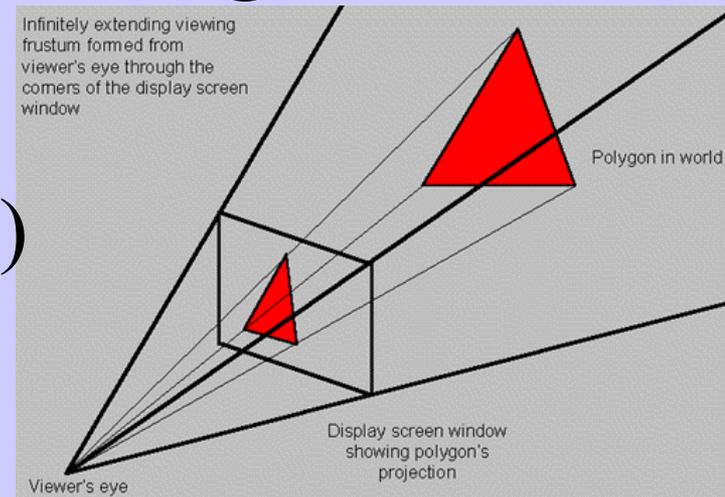


The APPLICATION stage

- 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

Model & View Transform

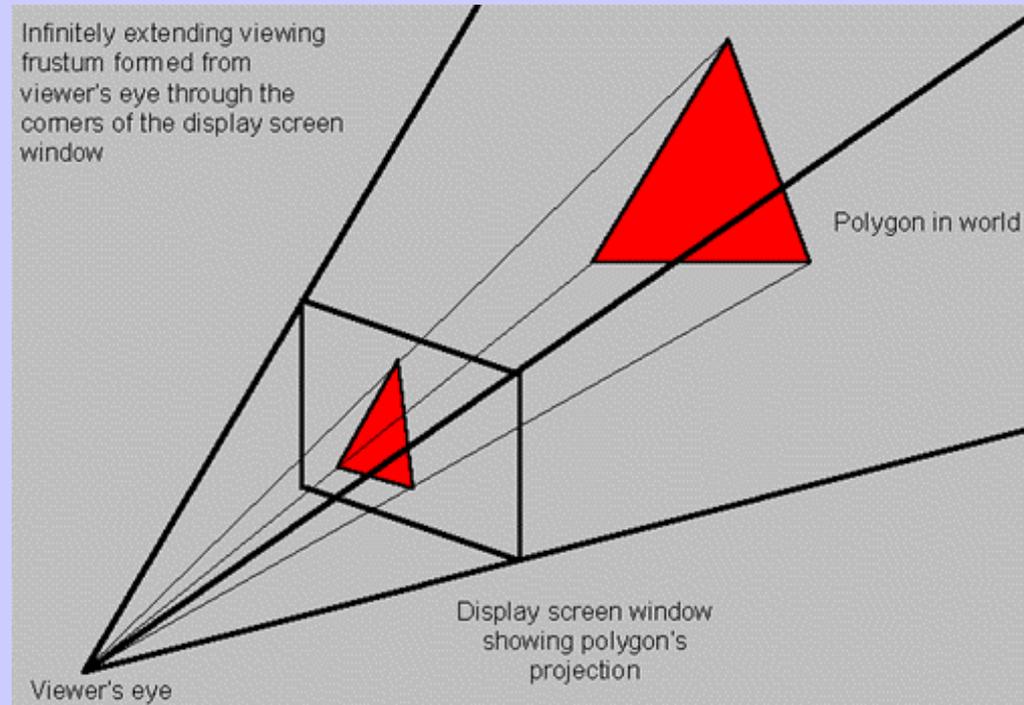
Vertex Shading

Projection

Clipping

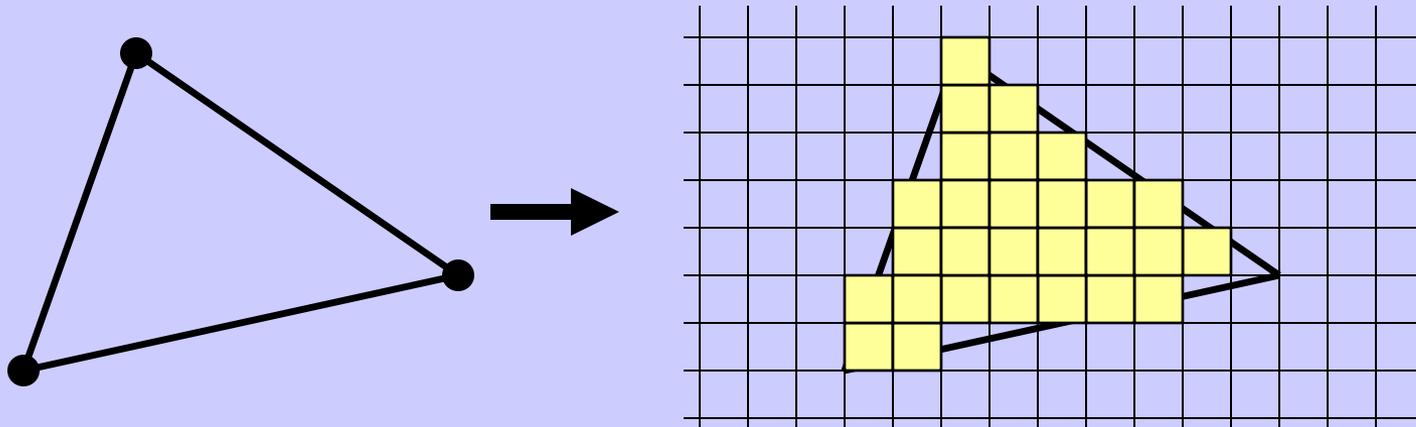
Screen Mapping

- (Instances)
- Vertex Shader
 - A program executed per vertex
 - Transformations
 - Projection
 - E.g., color per vertex
- Clipping
- Screen Mapping



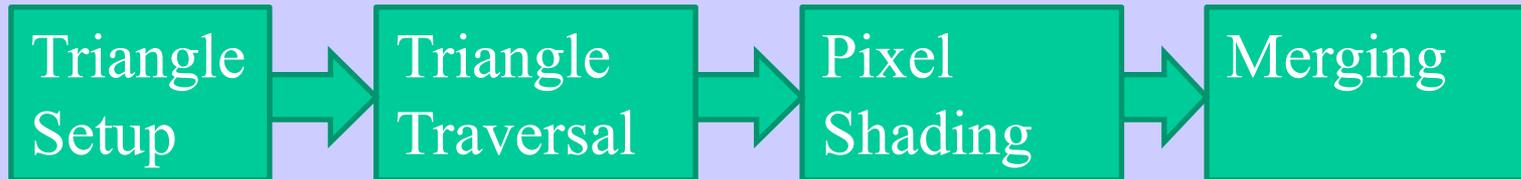
The RASTERIZER stage

- 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: sorts the primitives in the z-direction

The rasterizer stage



Triangle Setup:

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

Triangle Traversal

- Scan conversion

Pixel Shading

- Compute pixel color

Merging:

- output color to screen

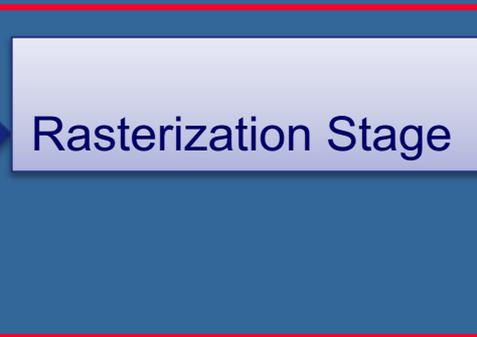
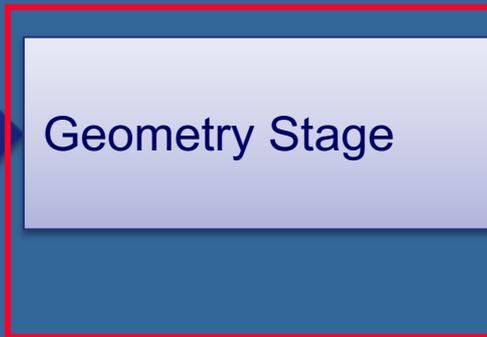
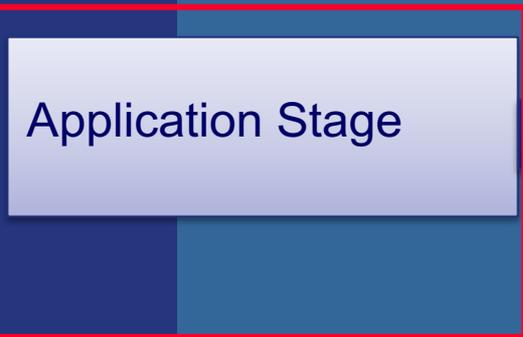
The three stages' correlation to hardware

The Application stage, geometry stage, and rasterizer stage

Rendering Pipeline and Hardware

CPU

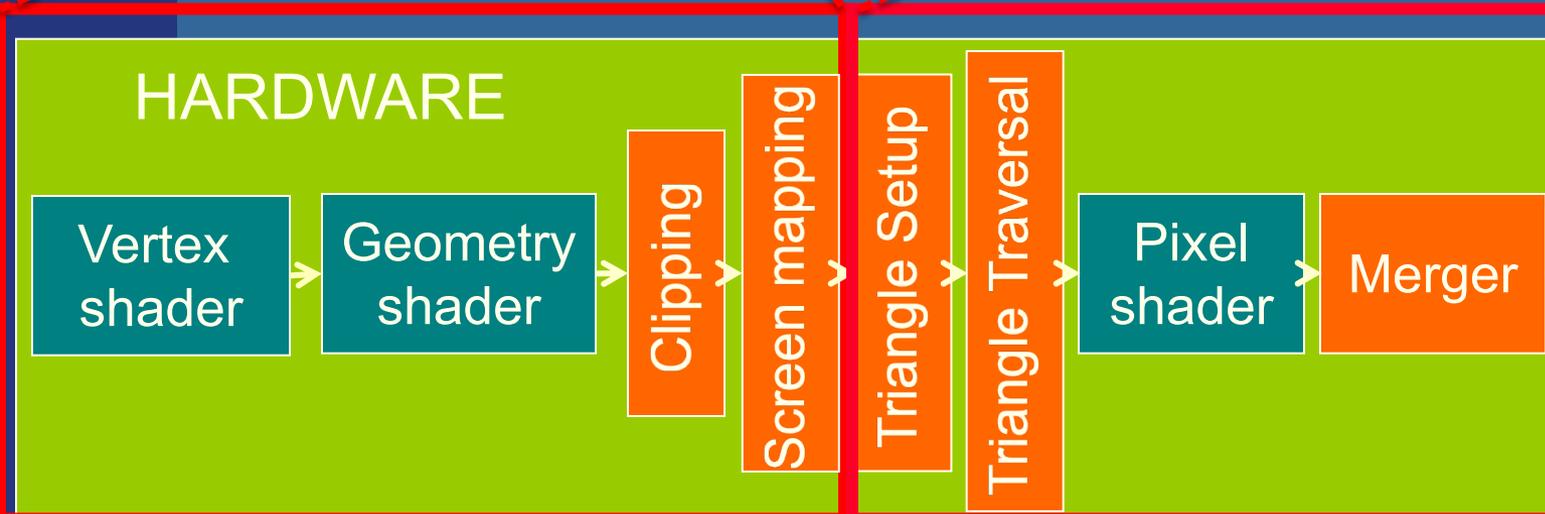
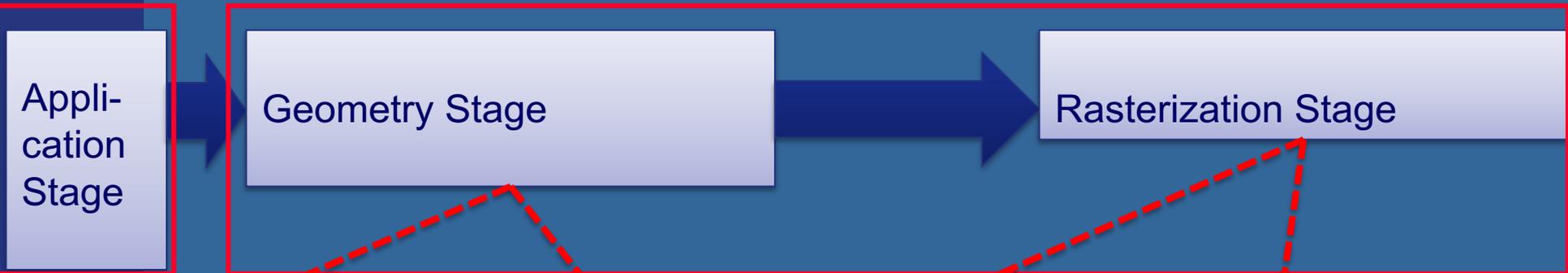
GPU



Rendering Pipeline and Hardware

CPU

GPU

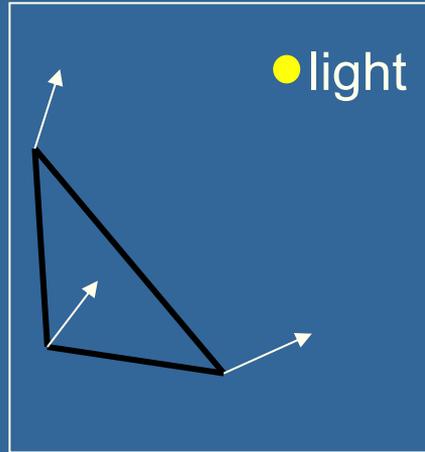
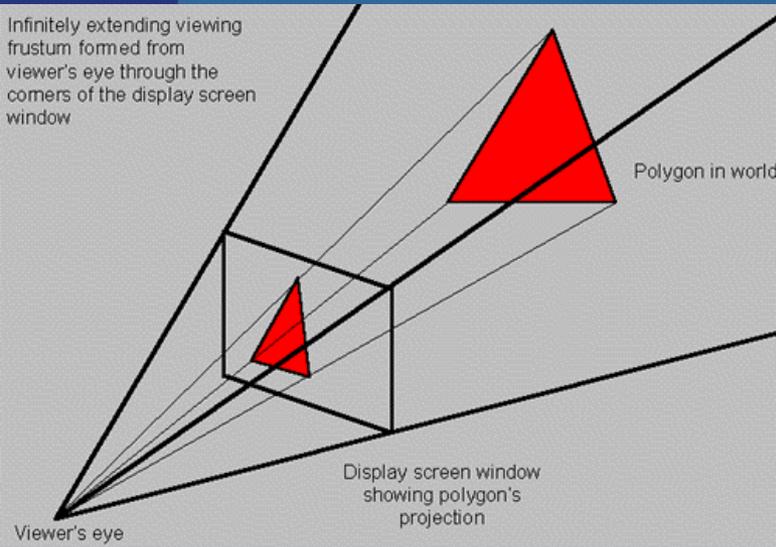


Display

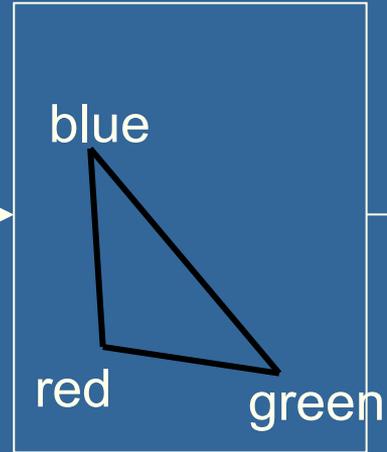
Hardware design

Vertex shader:

- Lighting (colors)
- Screen space positions



Geometry



Geometry Stage

Vertex
shader

Geometry
shader

Clipping

Screen mapping

Triangle Setup

Triangle Traversal

HARDWARE

Pixel
shader

Merger

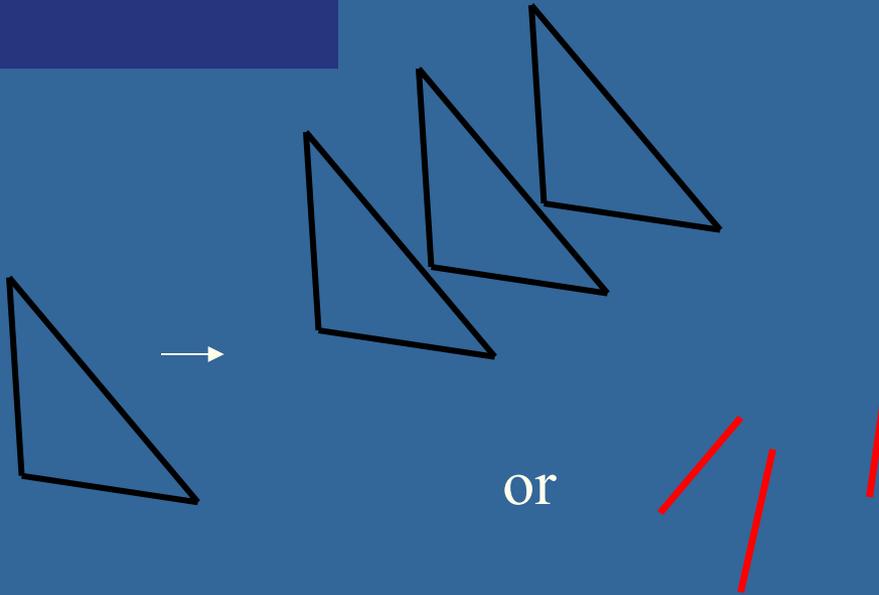
Display



Hardware design

Geometry shader:

- One input primitive
- Many output primitives



Geometry Stage

Vertex
shader

Geometry
shader

Clipping

Screen mapping

Triangle Setup

Triangle Traversal

HARDWARE

Pixel
shader

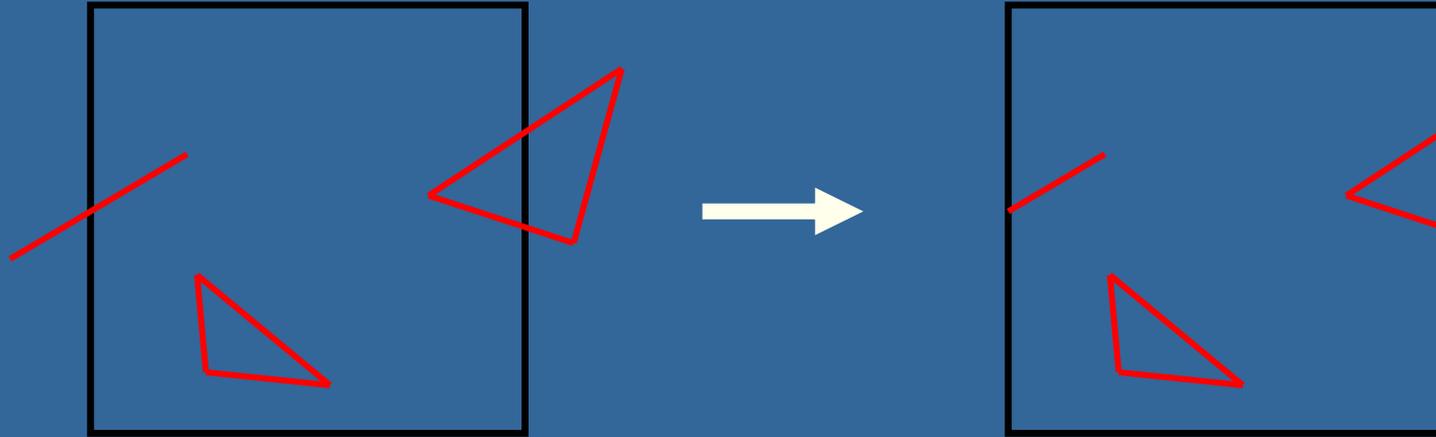
Merger

Display



Hardware design

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



Geometry Stage

Vertex
shader

Geometry
shader

Clipping

Screen mapping

Triangle Setup

Triangle Traversal

HARDWARE

Pixel
shader

Merger

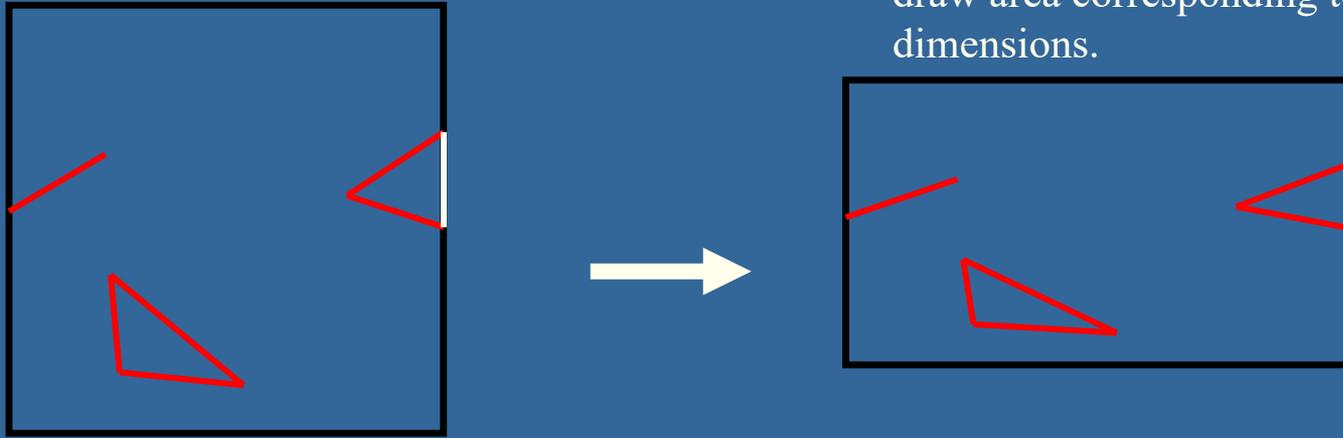
Display



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.



Geometry Stage

Vertex
shader

Geometry
shader

Clipping

Screen mapping

Triangle Setup

Triangle Traversal

HARDWARE

Pixel
shader

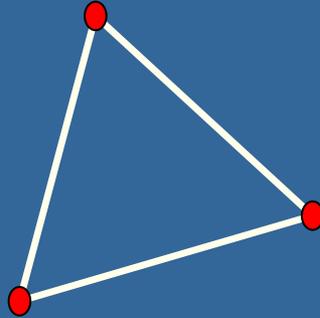
Merger

Display

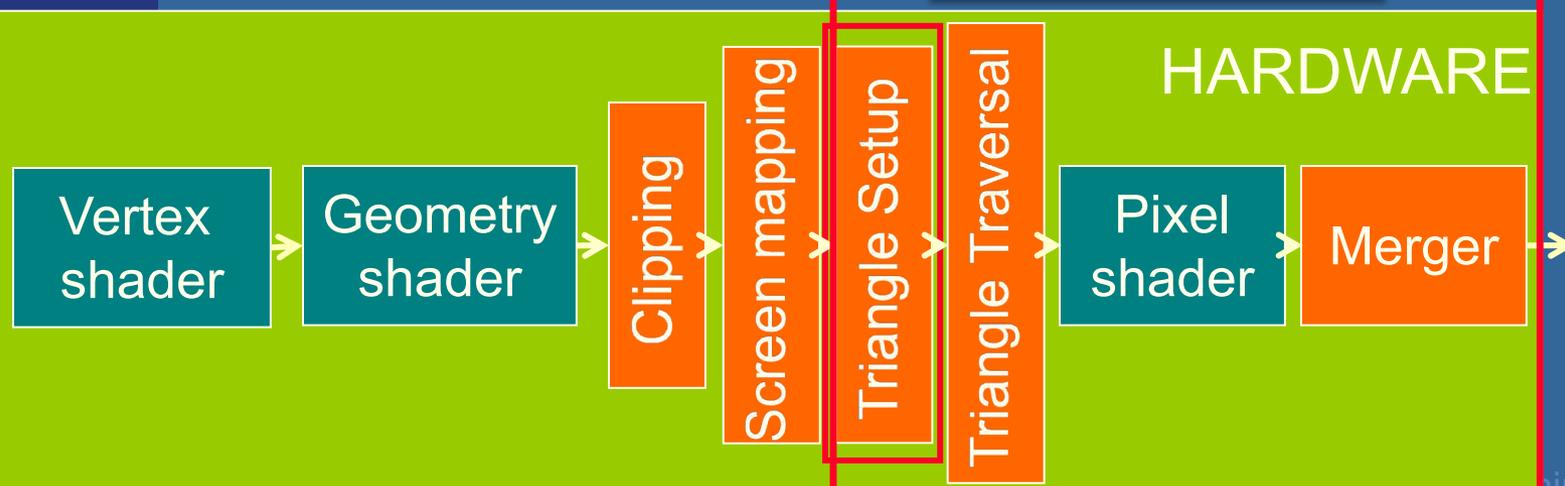


Hardware design

Collects three vertices into one triangle



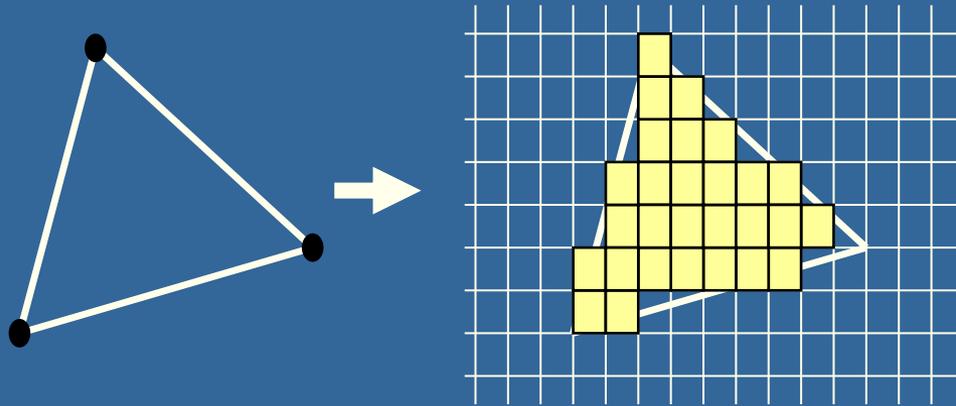
Rasterizer Stage



Display

Hardware design

Creates the fragments/pixels for the triangle



Rasterizer Stage

Vertex
shader

Geometry
shader

Clipping

Screen
mapping

Triangle
Setup

Triangle
Traversal

HARDWARE

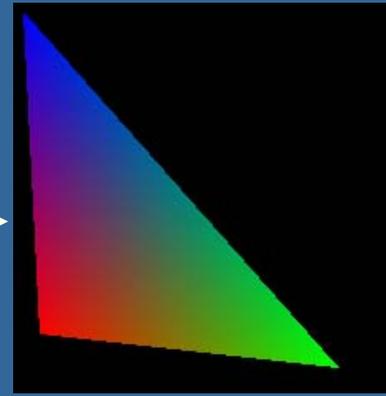
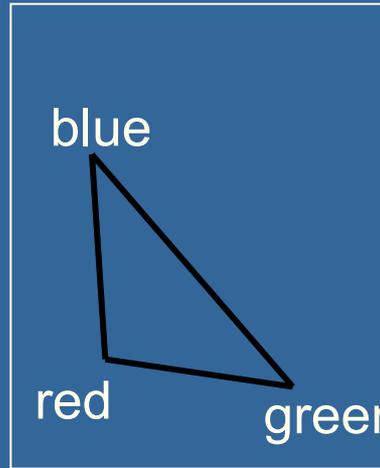
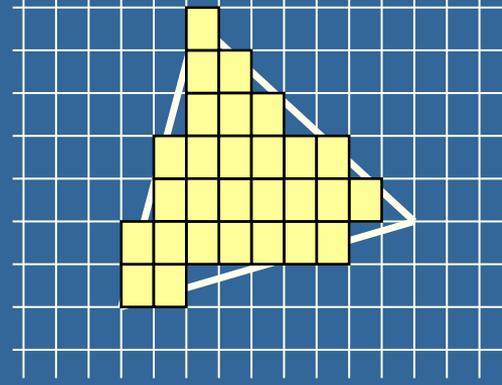
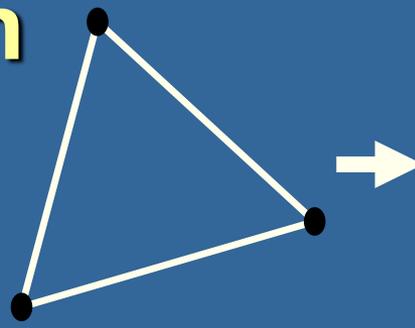
Pixel
shader

Merger

Display

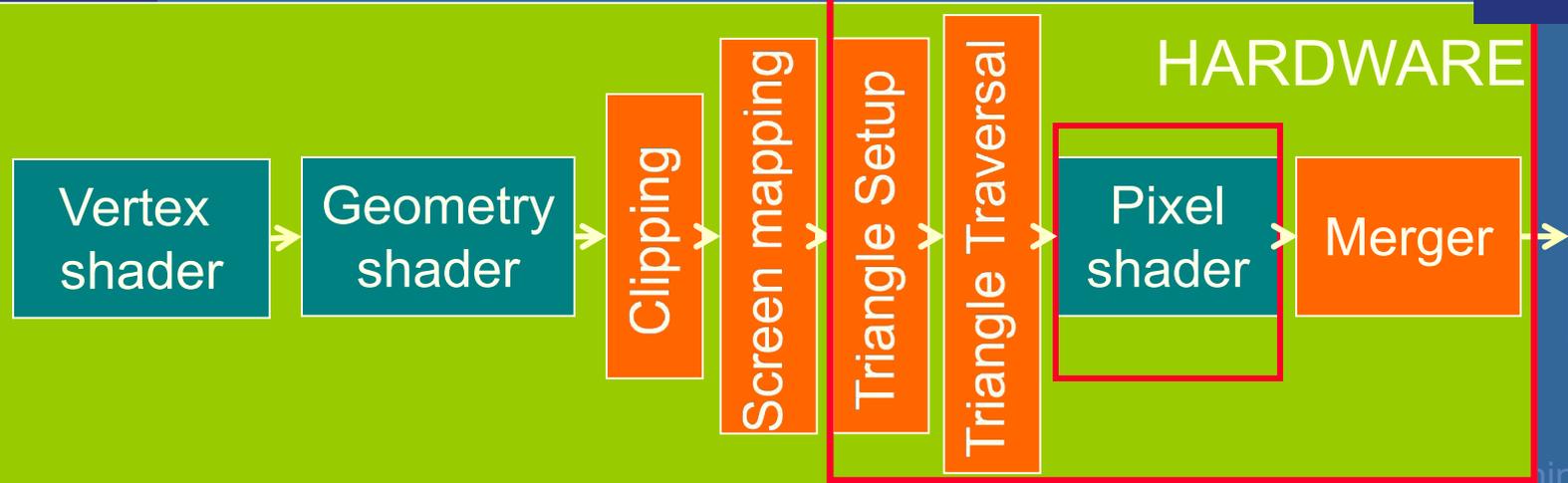


Hardware design



Rasterizer

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



Display

Hardware design

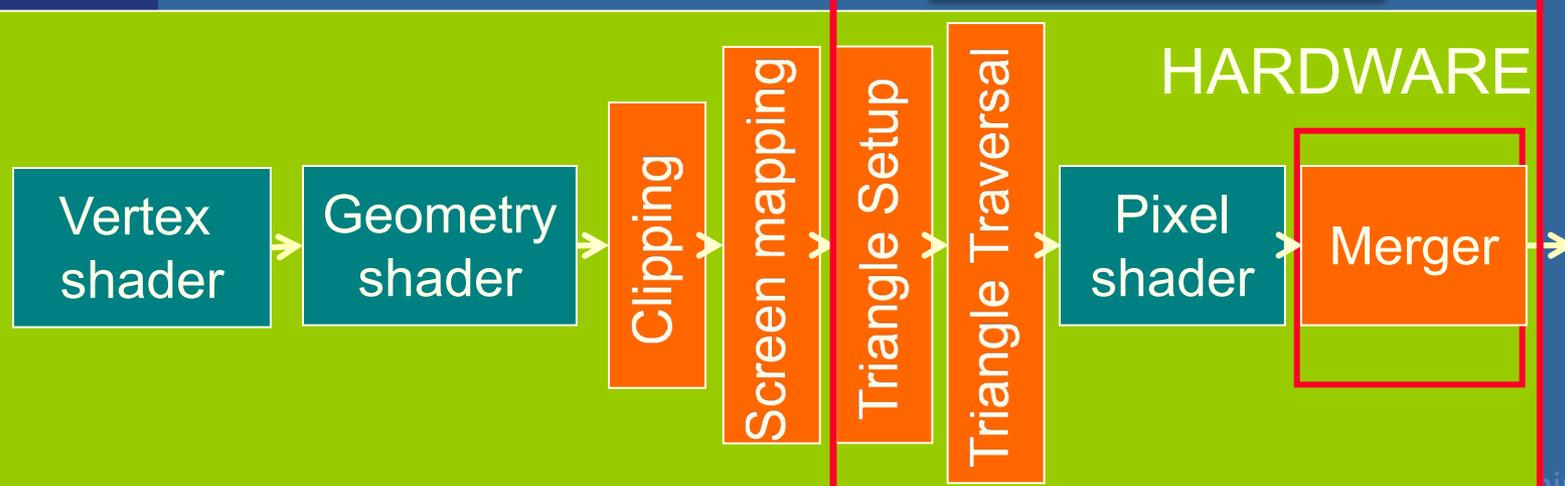
The merge units update the frame buffer with the pixel's color



Frame buffer:

- Color buffers
- Depth buffer
- Stencil buffer

Rasterizer Stage



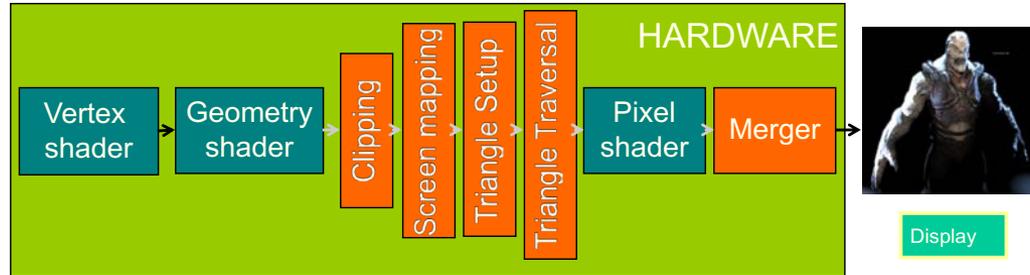
Display

Graphics Pipelines

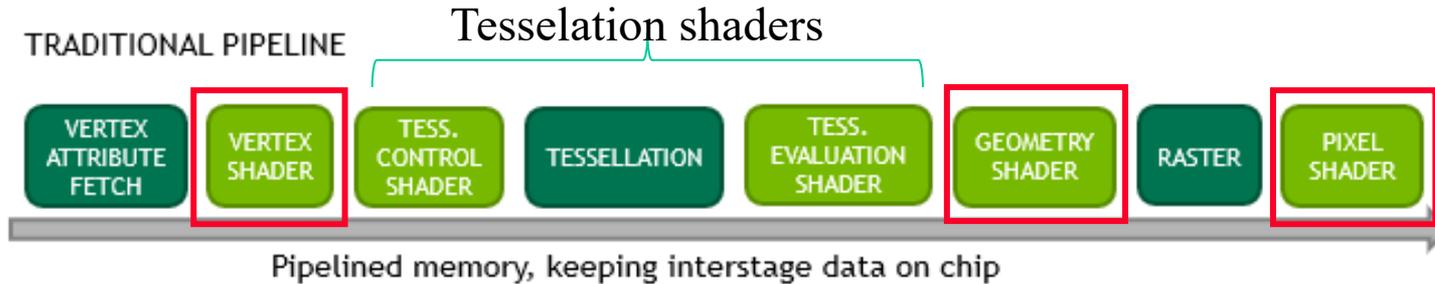
Compatibility:

- OpenGL 4.3
 - WebGL 2
 - OpenGL ES 3
- i.e., phones, web, PCs...

We focus on:

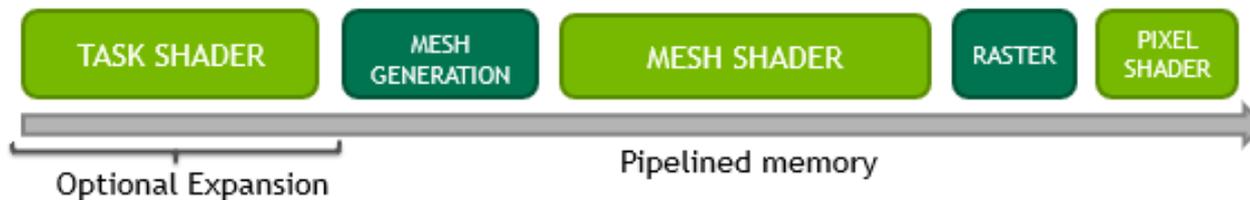


Full traditional pipeline:



Mesh shaders (still very new):

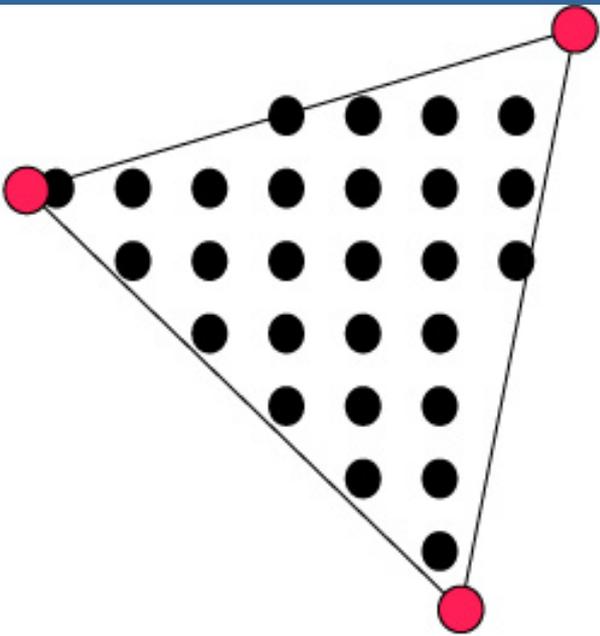
TASK/MESH PIPELINE



Compatibility:

- OpenGL 4.5 extension
- DirectX 12 Ultimate
- Vulkan

What is vertex and fragment (pixel) shaders?

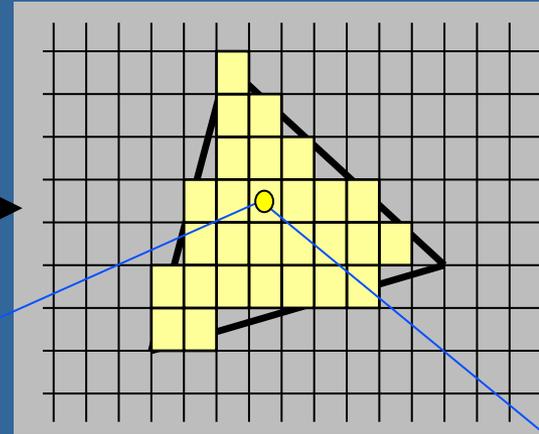
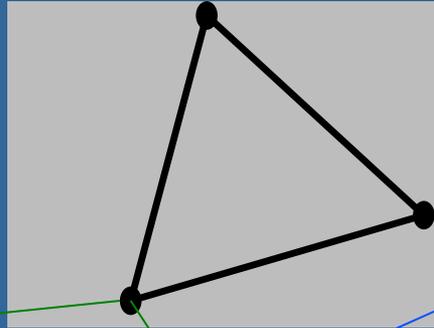
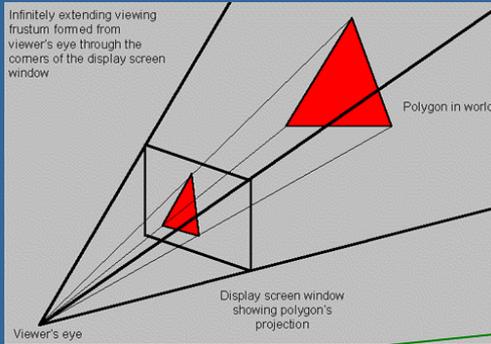


- Vertex shader: reads from textures
Writes output data per vertex, which are interpolated and input to each fragment shader invocation.
- Fragment shader: reads from textures, writes to pixel color
- Memory: Texture memory (read + write) typically 4 GB – 16 GB
- Program size: the smaller the faster

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

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

Shaders



```
// Vertex Shader
```

```
#version 420
```

```
layout(location = 0) in vec3 vertex;
```

```
layout(location = 1) in vec3 color;
```

```
out vec3 outColor;
```

```
uniform mat4 modelViewProjectionMatrix;
```

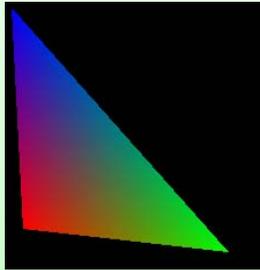
```
void main()
```

```
{
```

```
    gl_Position = modelViewProjectionMatrix *  
                  vec4(vertex,1);
```

```
    outColor = color;
```

```
}
```



```
// Fragment Shader:
```

```
#version 420
```

```
precision highp float;
```

```
in vec3 outColor;
```

```
layout(location = 0) out vec4 fragColor;
```

```
// Here, location=0 means that we draw to  
frameBuffer[0], i.e., the screen.
```

```
void main()
```

```
{
```

```
    fragColor = vec4(outColor,1);
```

```
}
```

Shaders

Example of a more advanced fragment shader:

```
precision highp float;

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

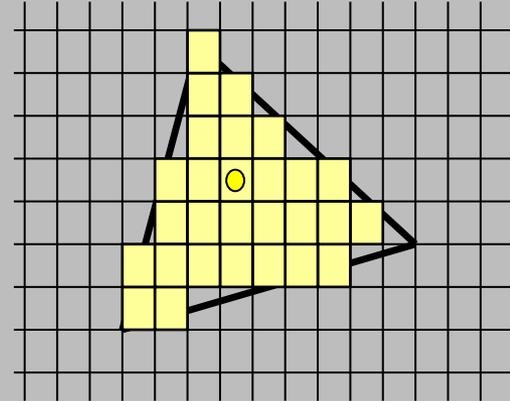
uniform float val;

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++:

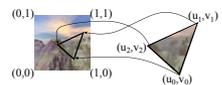
```
float positions[] = {
// X      Y      Z      per vertex
  0.0f,   0.5f,  1.0f, // v0
 -0.5f,  -0.5f,  1.0f, // v1
  0.5f,  -0.5f,  1.0f // v2
  ...
};
```

// 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
  ...
};
```

float normals[] = {...};

float textureCoords[] = {...};



OpenGL:

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; // Per-pixel-interpolated value
```

```
void main()
```

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

CPU-side

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



C++:

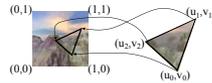
```
float positions[] = {
// X      Y      Z
  0.0f,   0.5f,  1.0f, // v0
 -0.5f,  -0.5f,  1.0f, // v1
  0.5f,  -0.5f,  1.0f  // v2
```



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

```
0.0T,   0.0T,  1.0T // c2
...      // ... cn
};
```

```
float normals[] = {...};
float textureCoords[] = {...};
```



OpenGL:

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.
```

```
glBindBuffer( GL_ARRAY_BUFFER, positionBuffer );
// Attach positionBuffer to vertexArrayObject,
// in location 0. 3 floats per vertex
glVertexAttribPointer(0, 3, GL_FLOAT, ...);
```

```
// Makes colorBuffer the current array buffer for subsequent commands.
```

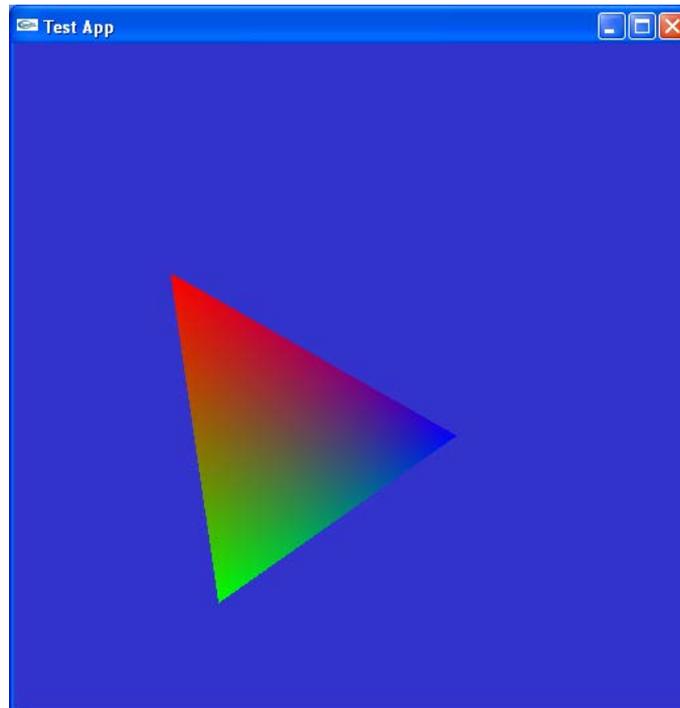
```
glBindBuffer( GL_ARRAY_BUFFER, colorBuffer );
// Attaches colorBuffer to vertexArrayObject,
// in location 1. 3 floats per vertex
glVertexAttribPointer(1, 3, GL_FLOAT, ... );
```

```
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



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

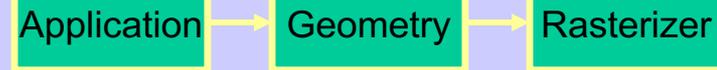
```
class GfxObject {
public:
    load("filename"); // Creates m_shaderProgram + m_vertexArrayObject
    render()
    {
        /* You may want to initiate more OpenGL states, e.g., for
           textures (more on that in further lectures) */
        glUseProgram(m_shaderProgram);
        glBindVertexArray(m_vertexArrayObject);
        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 primitives 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

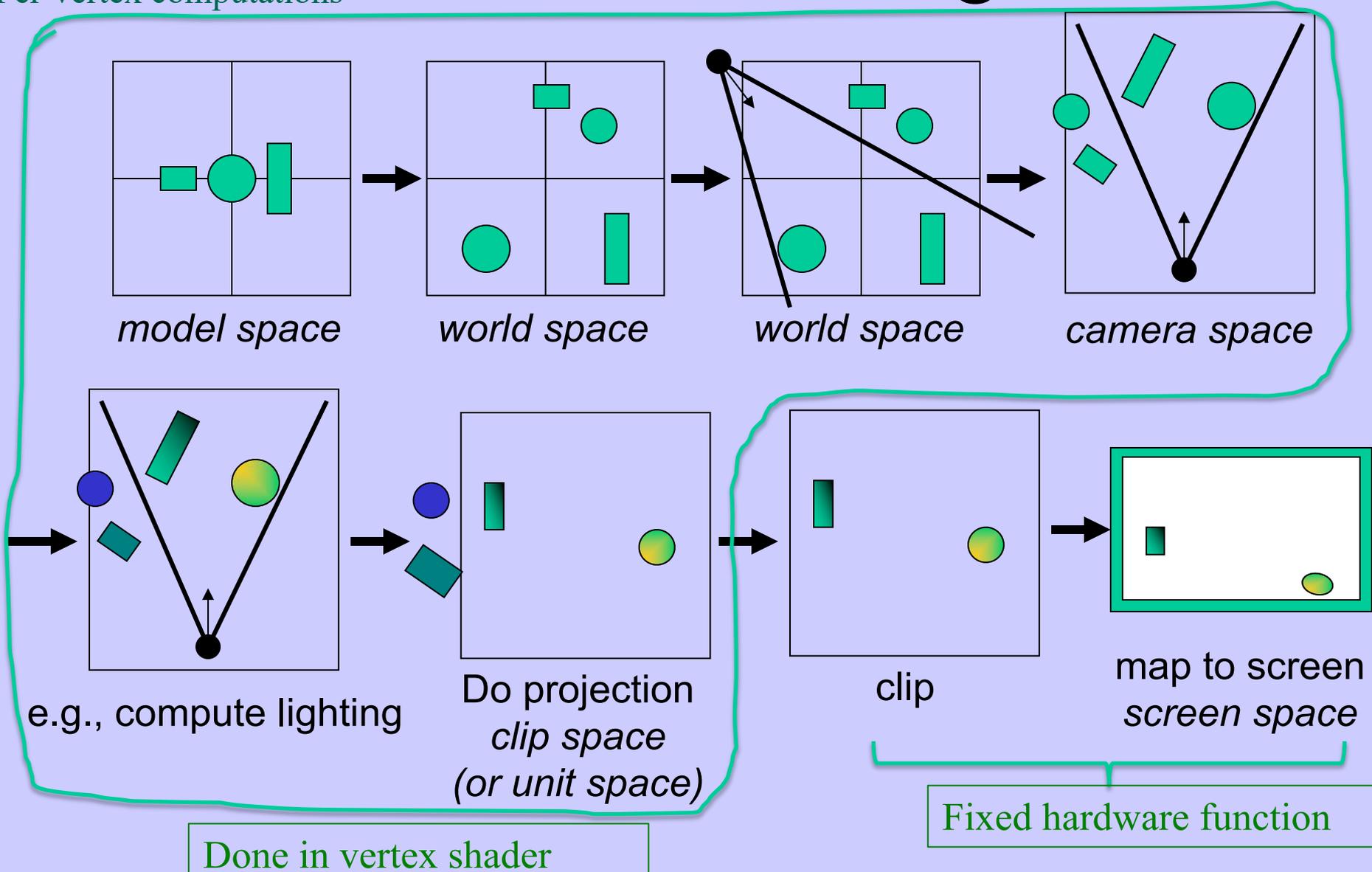
Application

Geometry

Rasterizer

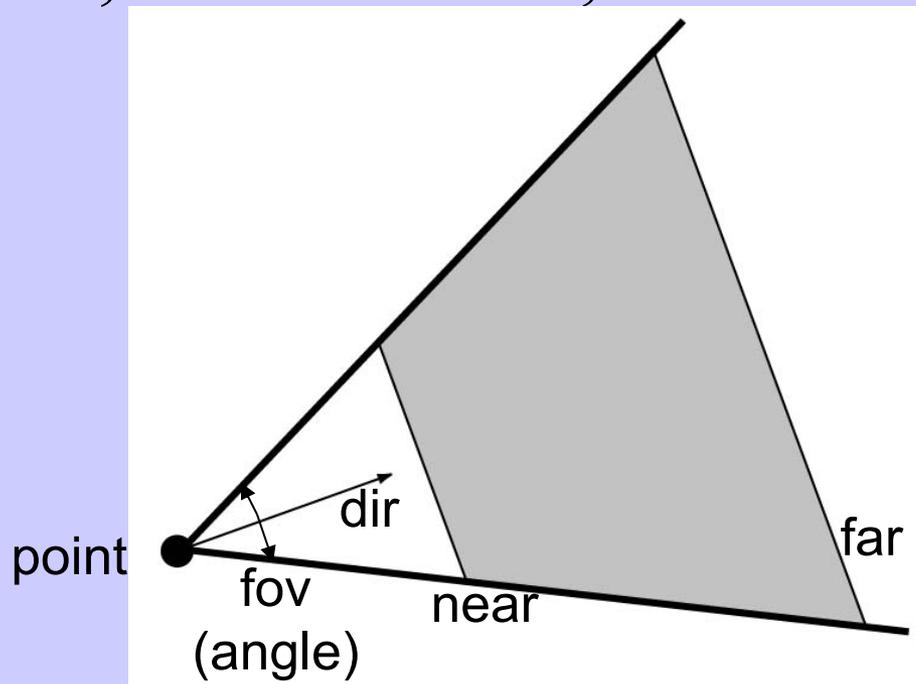
GEOMETRY Stage

Per-vertex computations



Virtual Camera

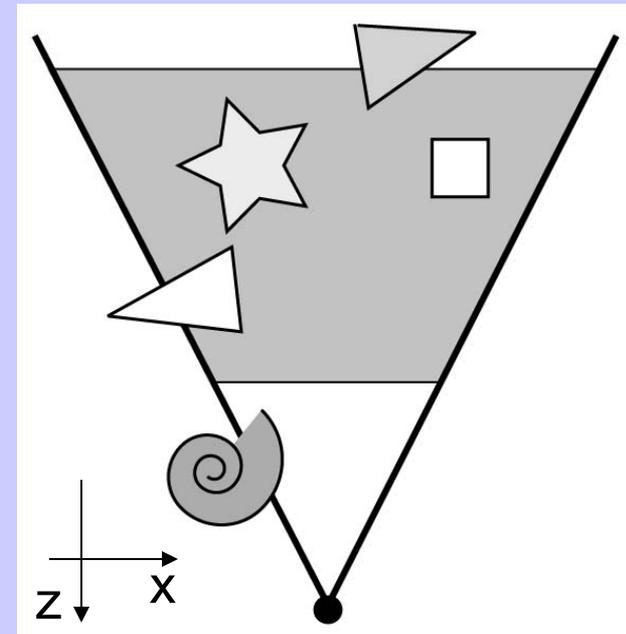
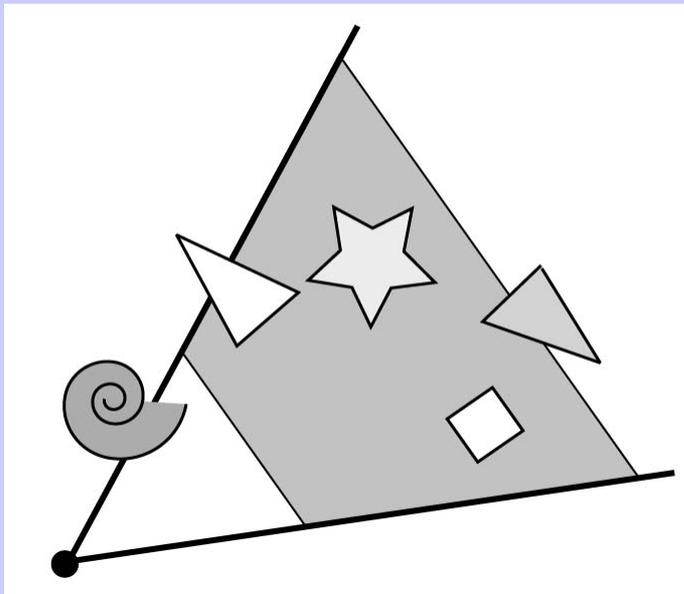
- Defined by position, direction vector, up vector, field of view, near and far plane.



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

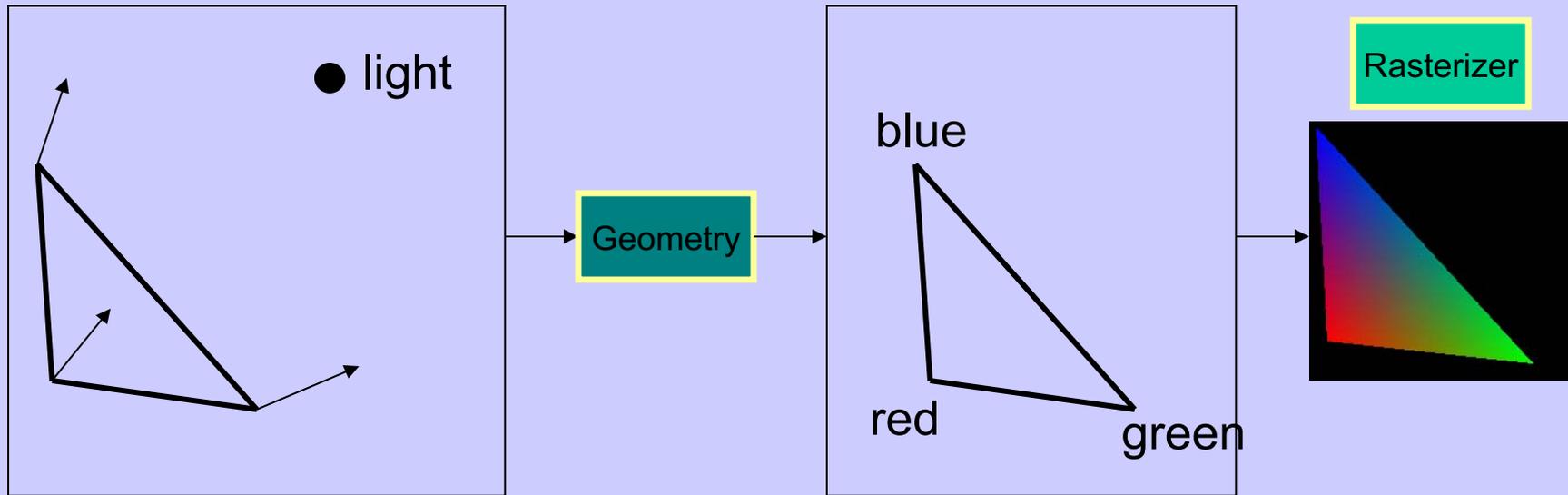
GEOMETRY - The view transform

- 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 - Lighting

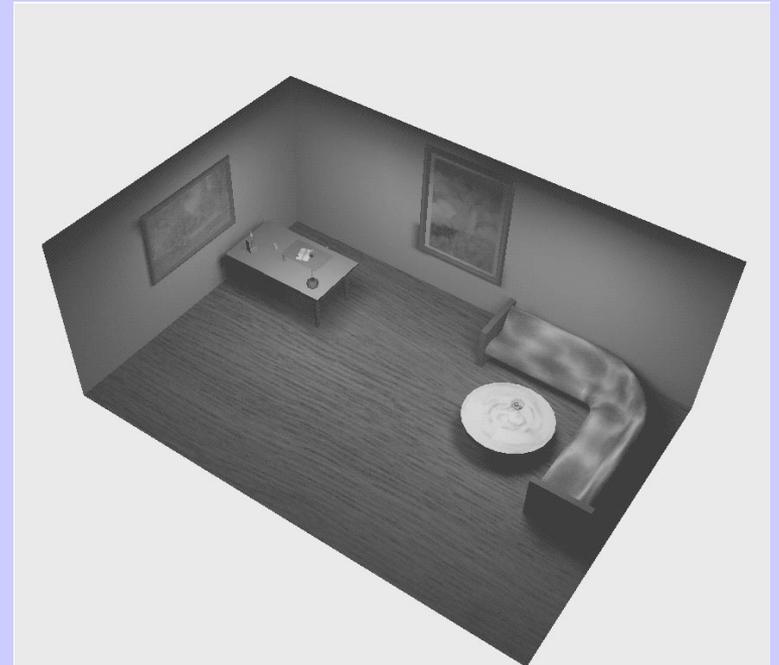
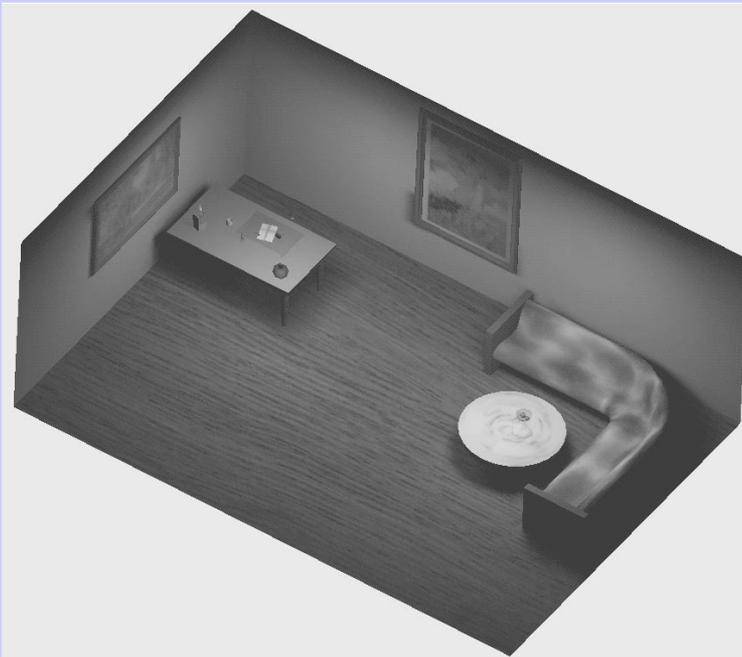
- Compute "lighting" at vertices



- Try to mimic how light in nature behaves
 - Hard so uses empirical models, hacks, and some real theory
- Much more about this in later lecture

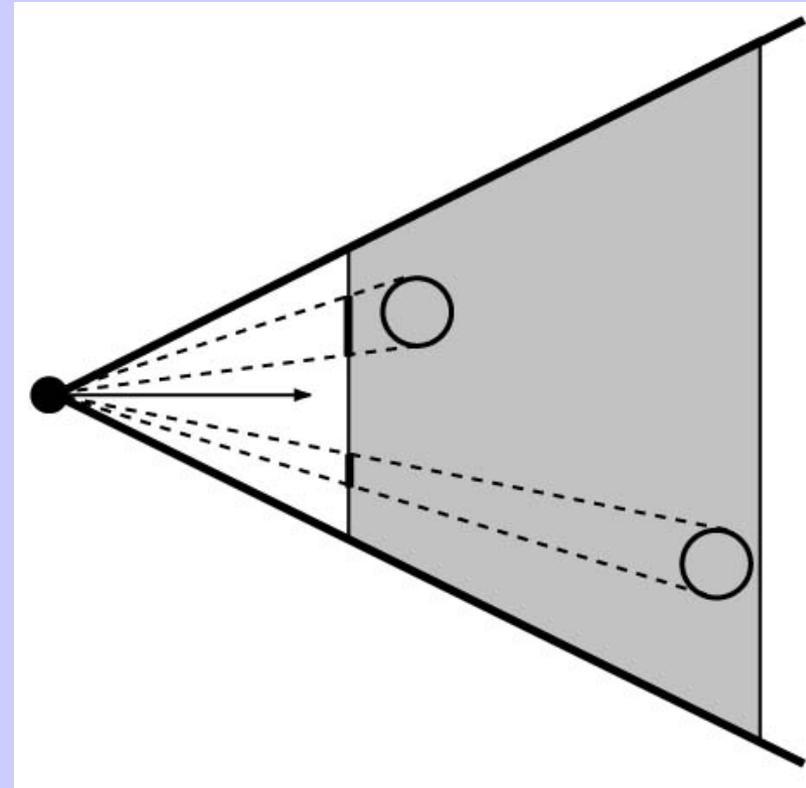
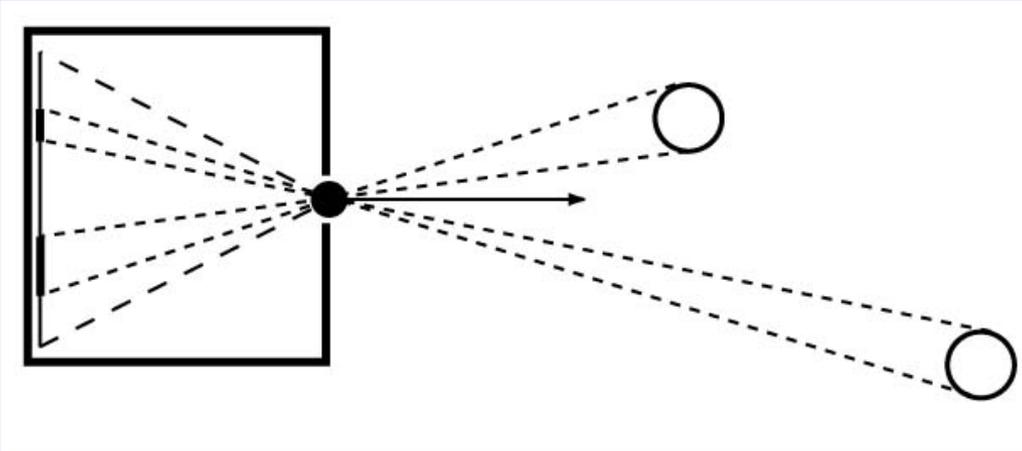
GEOMETRY - Projection

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



GEOMETRY - Projection

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



GEOMETRY

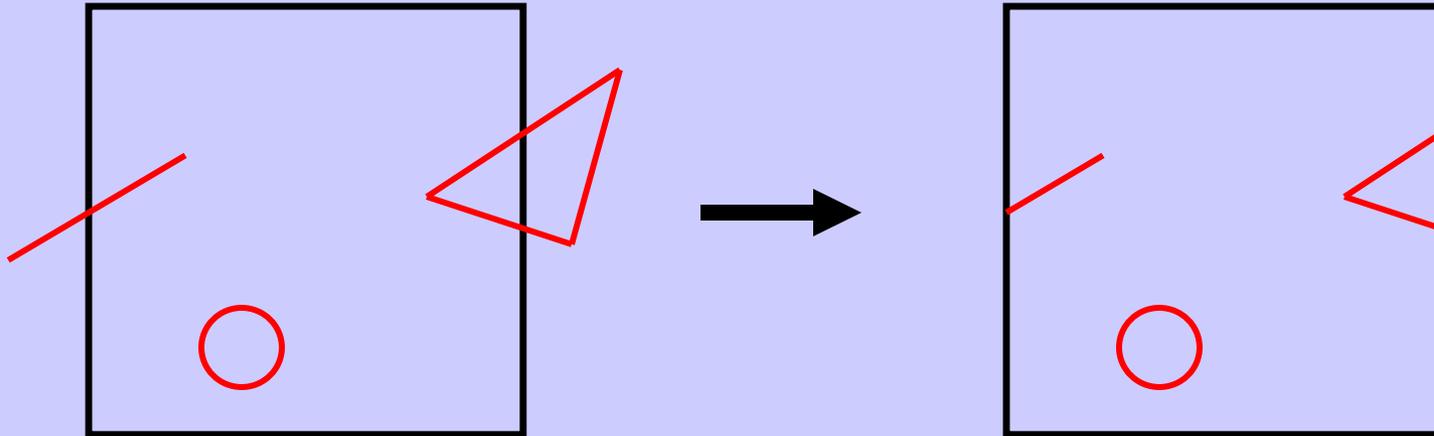
Application

Geometry

Rasterizer

Clipping and Screen Mapping

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



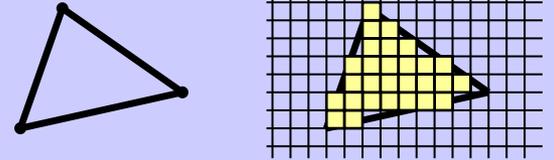
- Screen mapping, scales and translates the square so that it ends up in a rendering window
- These "screen space coordinates" together with Z (depth) are sent to the rasterizer stage

The RASTERIZER

in more detail

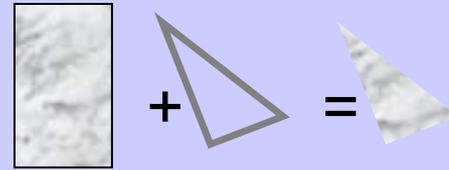
- Scan-conversion

- Find out which pixels are inside the primitive

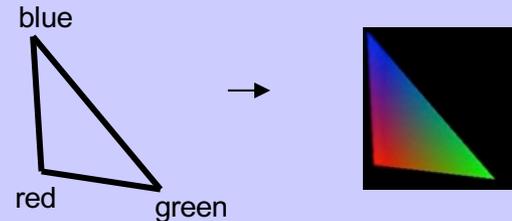


- Fragment shaders

- E.g. put textures on triangles



- Use interpolated data over triangle
- and/or compute per-pixel lighting



- Z-buffering

- Make sure that what is visible from the camera really is displayed

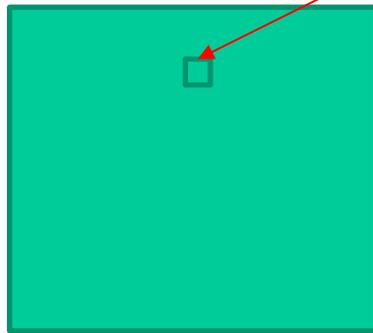
- Doublebuffering



The default frame buffer:

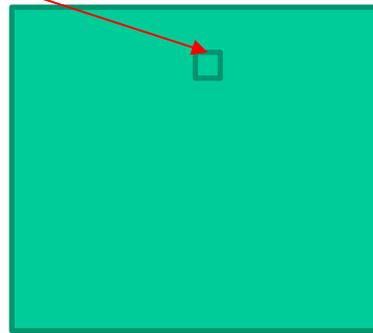
Typically: Front + Back color buffers + Z buffer + (Stencil buffer)

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



Front buffer
(rgb color buffer)

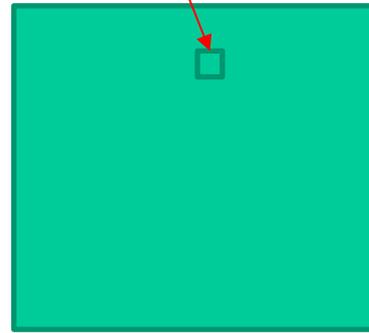
Last fully finished
drawn frame.
Is displayed.



Back buffer
(rgb color buffer)

Color buffer we draw to.
Not displayed yet.

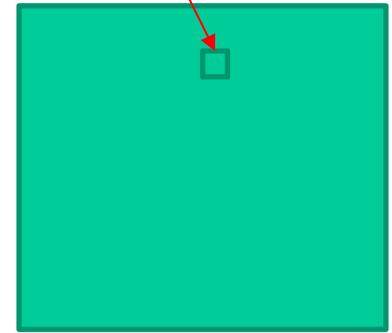
Stores fragment's
depth value per
pixel, typically: (16),
24, or 32 bits.



Z buffer
(depth)

To resolve visibility

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



Stencil buffer

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

The RASTERIZER

Application

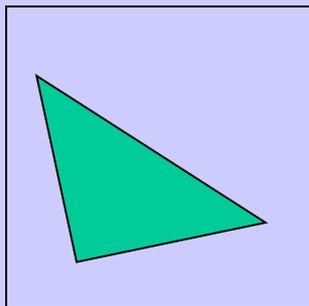
Geometry

Rasterizer

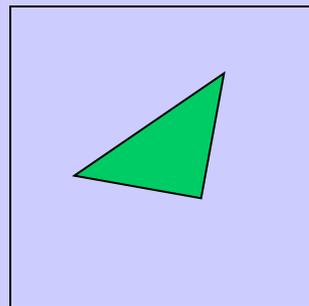


Z-buffering

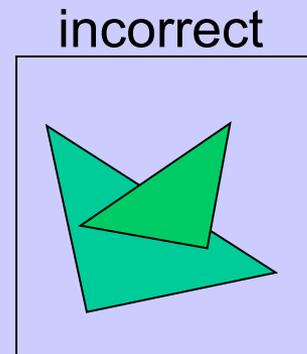
- A triangle that is covered by a more closely located triangle should not be visible
- Assume two equally large tris at different depths



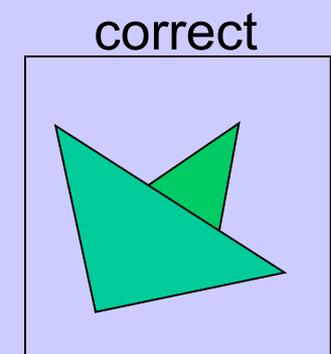
Triangle 1



Triangle 2



Draw 1 then 2



Draw 2 then 1

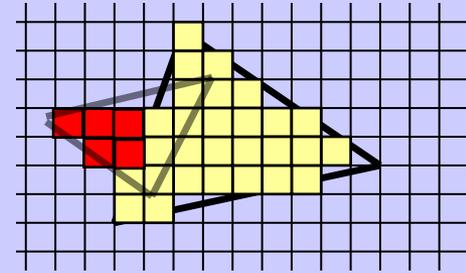
The RASTERIZER

Application

Geometry

Rasterizer

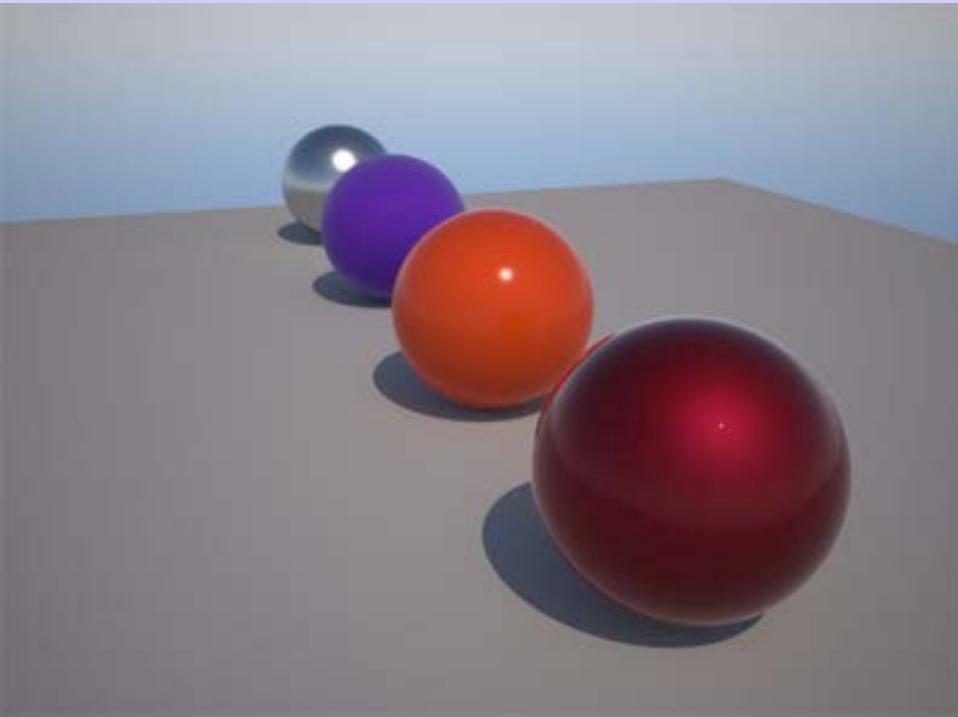
Z-buffering



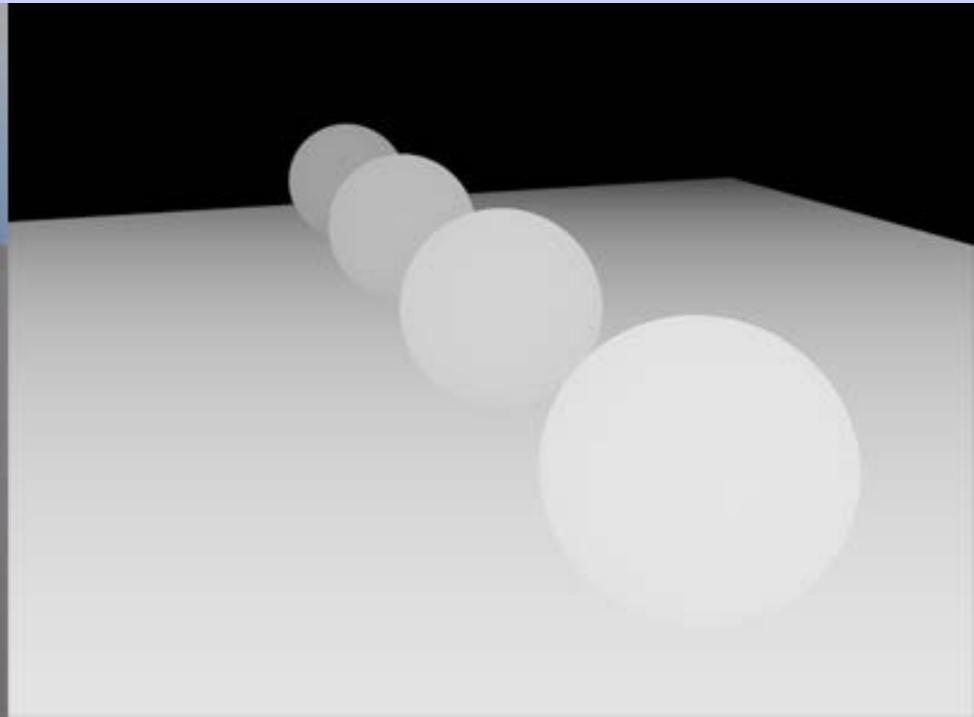
- Would be nice to avoid sorting...
- The Z-buffer (aka depth buffer) solves this
- Idea:
 - Store z (depth) at each pixel
 - When rasterizing a triangle, compute z at each pixel on triangle
 - Compare triangle's z to Z-buffer z -value
 - If triangle's z is smaller, then replace Z-buffer and color buffer
 - Else do nothing
- Can render in any order

The RASTERIZER

Z-buffer



The color buffer



The z-buffer
(or depth buffer)

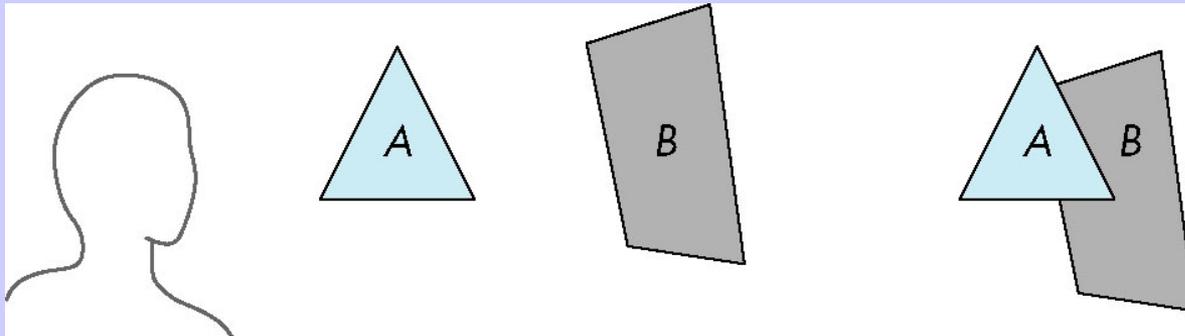
The RASTERIZER

Z-buffer



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.

Z-Buffer Algorithm

- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each new fragment, d_{new} , to depth in z buffer, d_{zb}
- If $d_{new} < d_{zb}$ (new fragment is closer to cam), replace pixel's color and z-buffer value.

The RASTERIZER

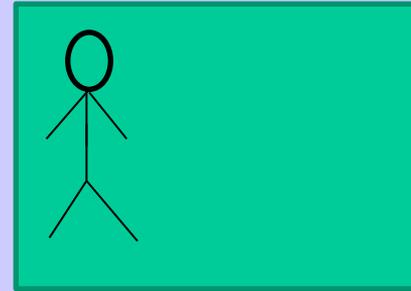
double-buffering

Application

Geometry

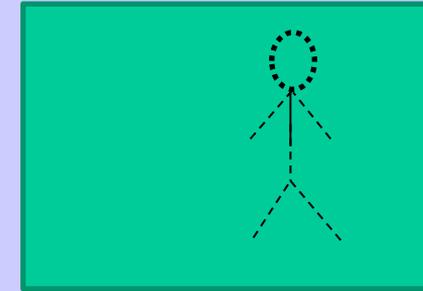
Rasterizer

- We do not want to show the image until its drawing is finished.



Front buffer
(rgb color buffer)

Last fully finished
drawn frame.



Back buffer
(rgb color buffer)

Color buffer we draw to.
Not displayed yet.

- 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 vsynch or screen tearing will occur...
i.e., when the swap happens in the middle of the screen with respect to the screen refresh rate.

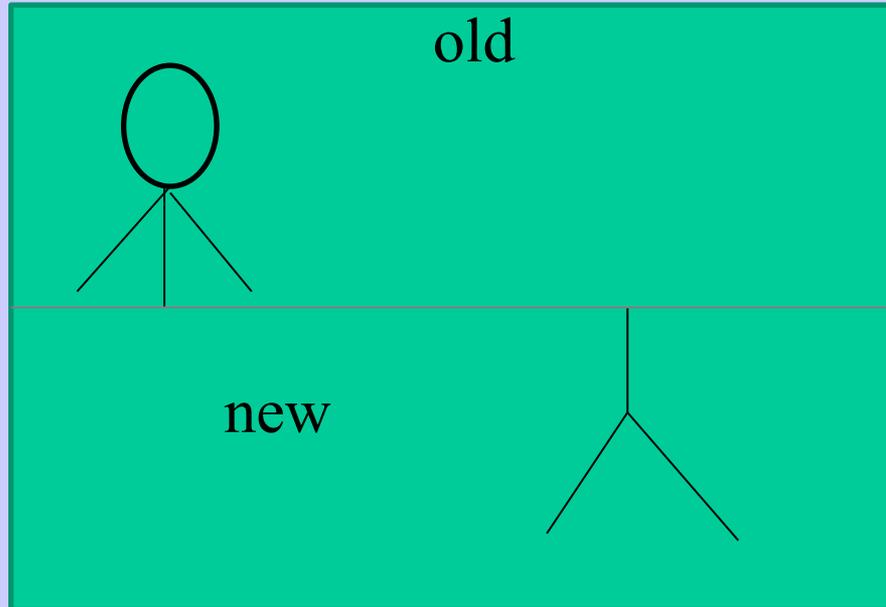
The RASTERIZER

Application

Geometry

Rasterizer

double-buffering – screen tearing



Example if the swap happens here (w.r.t the screen refresh rate).

Screen Tearing

Swapping
back/front buffers



Screen tearing is solved by using V-Sync.

vblank →

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 intervals (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.

What is important:

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

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();
}
```

