

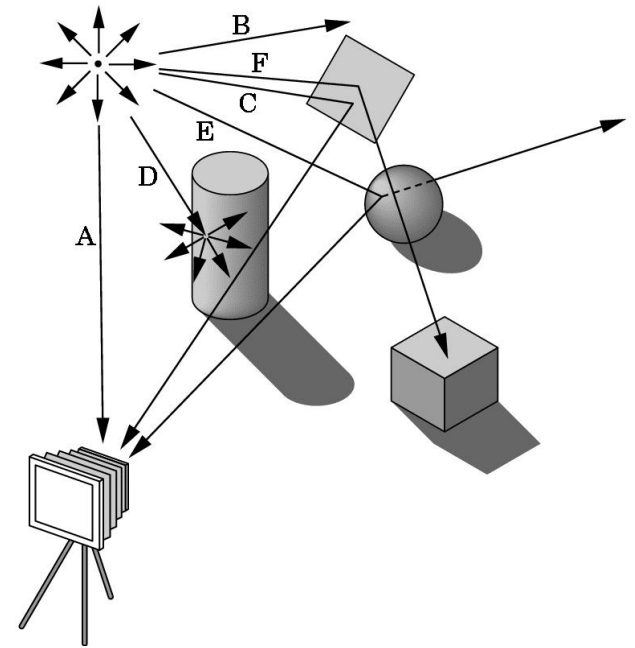
# TDA361 - Computer Graphics



Ulf Assarsson  
Department of Computer Engineering  
Chalmers University of Technology

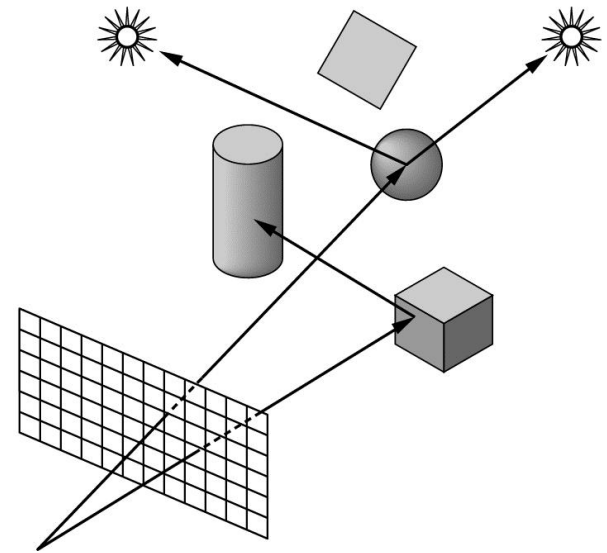
# Tracing Photons

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 slow



# I'm only here to help...

1. I am located in room 4115 in "EDIT-huset"
2. Email: uffe at chalmers dot se
3. Phone: 031-772 1775 (office)
4. Course assistant:
  1. ola.olsson at chalmers dot se
  2. billeter at chalmers dot se (Markus Billeter)
  3. maria.lemon at hotmail dot com

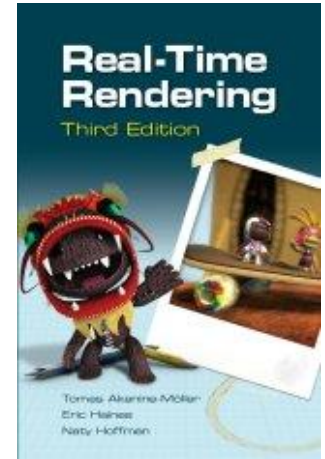
# Studentrepresentanternas ansvar

- Informerar sig om sina kurskamraters synpunkter på kursen.
- Vidarebefordrar dessa samt deltar i övrigt i diskussionen vid mötena med egna synpunkter.
- Kan föreslå kursspecifika frågor i kursenkäten.
- Informerar sina kurskamrater om diskussioner och rekommendationer från mötena.

Ersättning utgår i form av presentkort på 200 kr på Cremona.

# Course Info

- Real Time Rendering, 3<sup>rd</sup> edition
  - Available on Cremona
- Homepage:
  - Google “TDA361” or
  - “Computer Graphics Chalmers”



This is a screenshot of a web browser displaying the course homepage for 'TDA361/DIT220 - Computer graphics 2011 lp2'. The page has a white background with a blue header bar containing the course title and navigation links for 'Home', 'Schedule', 'Literature', 'Tutorials', and 'Exam'. Below the header, there is a 'Very important' notice about refreshing the page. A 'NOTE' section states that the course will be given in study period 2. A 'COURSE-FW' section provides details about the course start, contact information for the teacher and examiner, and a list of links. At the bottom, there is a 'More Links' section with various references and resources.

# Tutorials

- Rooms 4211,4213,4215,4220
  - Or your favorite place/home
- 4<sup>th</sup> floor EDIT-building
- EntranceCards (inpasseringskort)
  - Automatically activated for all of you that are course registered and have a CTH/GU-entrance card (inpasseringskort)
- Recommended to do the tutorials in groups (Labgrupper) of 2 and 2

# Overview of the Graphics Rendering Pipeline and OpenGL



CHALMERS

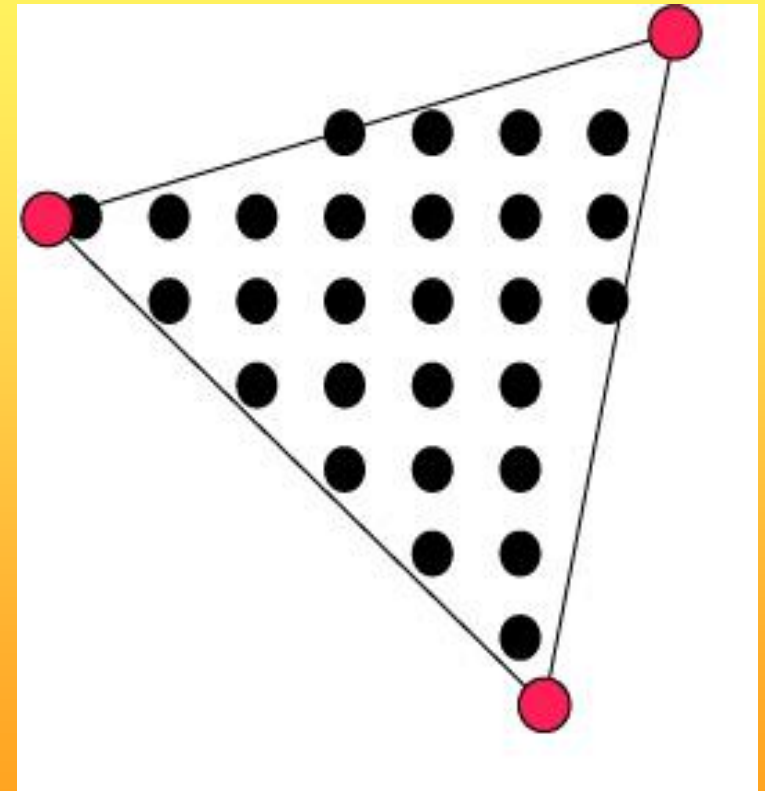
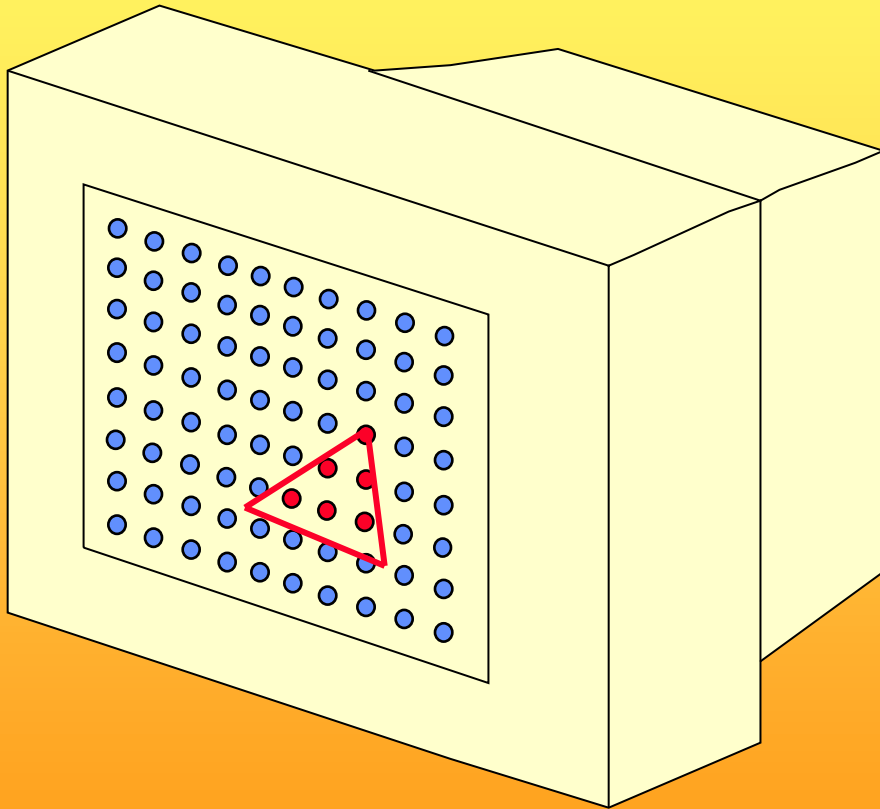
# 3D Graphics



Ulf

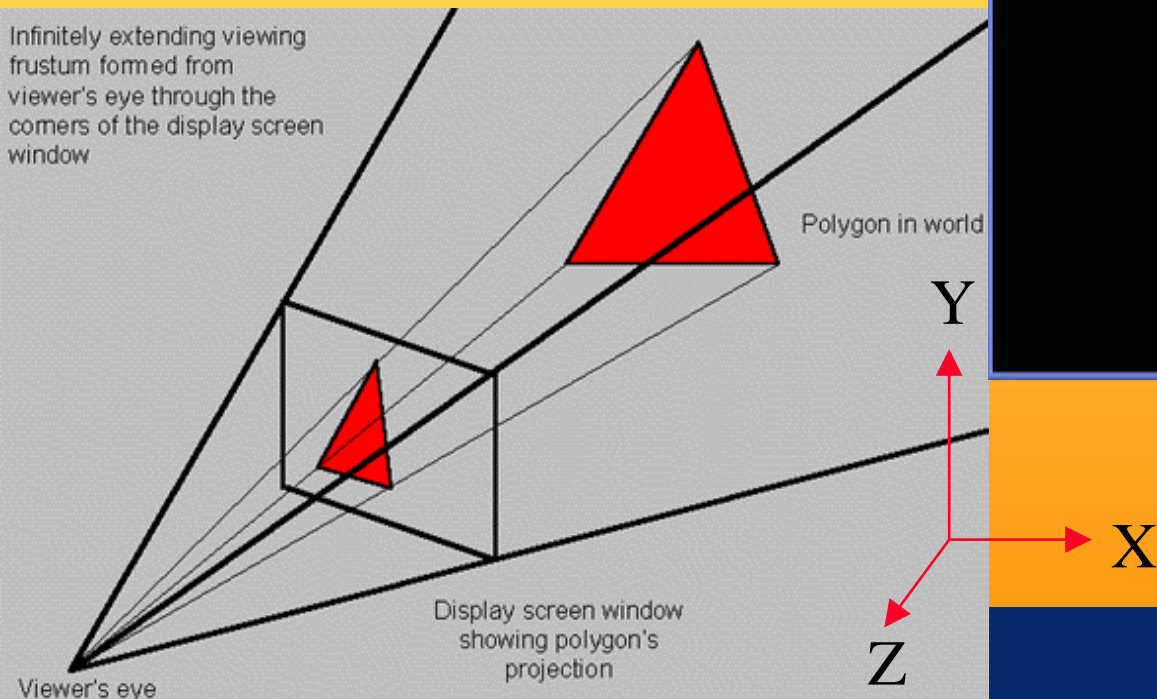
Assarsson

# The screen consists of many pixels

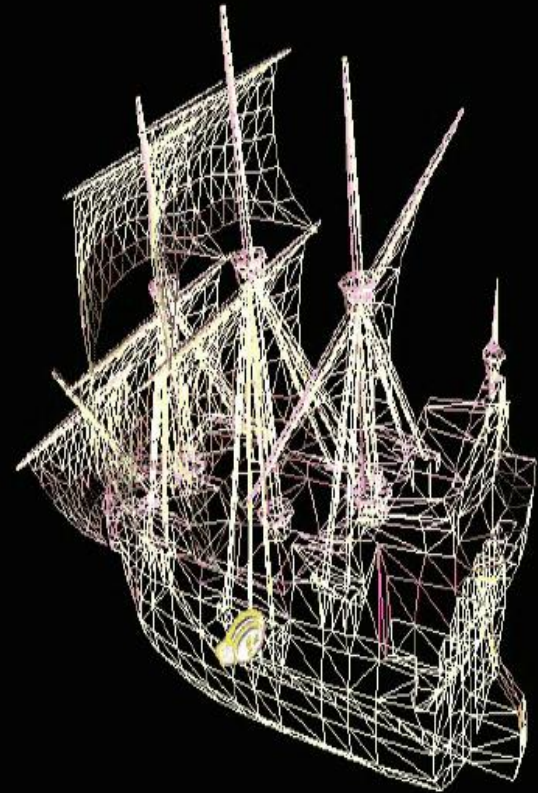


# 3D-Rendering

- Objects are often made of triangles
- $x, y, z$ - coordinate for each vertex



(C) 1998 Evans & Sutherland Glaze v3.1



# 4D Matrix Multiplication

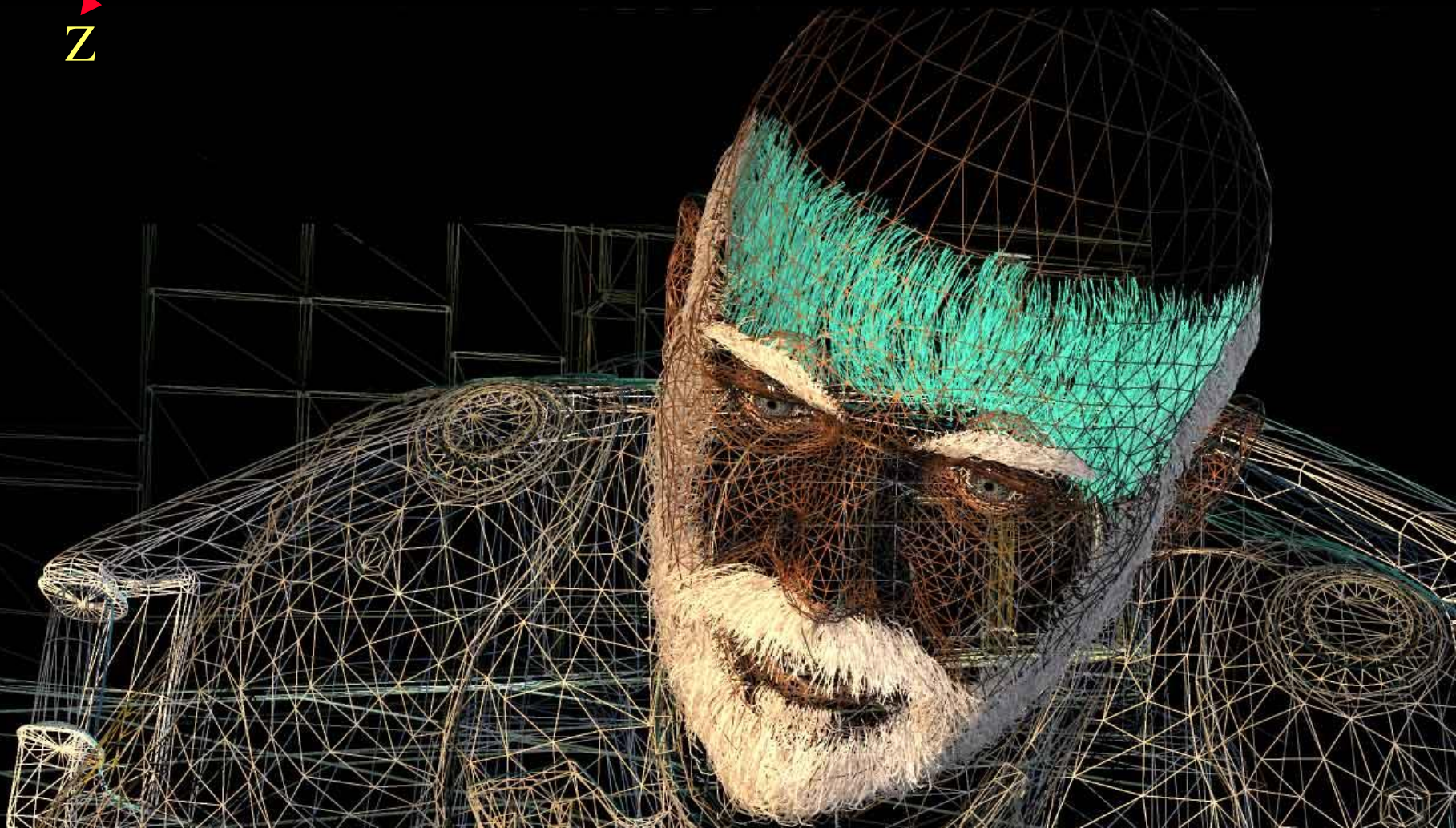
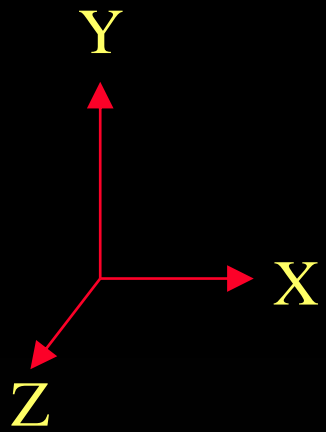
$$\begin{bmatrix} s_x & \bullet & \bullet & t_x \\ \bullet & s_y & \bullet & t_y \\ \bullet & \bullet & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$



# Real-Time Rendering

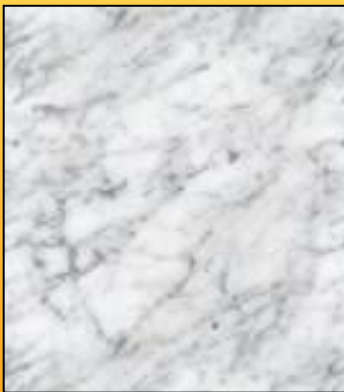






# Textures

- One application of texturing is to "glue" images onto geometrical object



+



=



# Texturing: Glue images onto geometrical objects

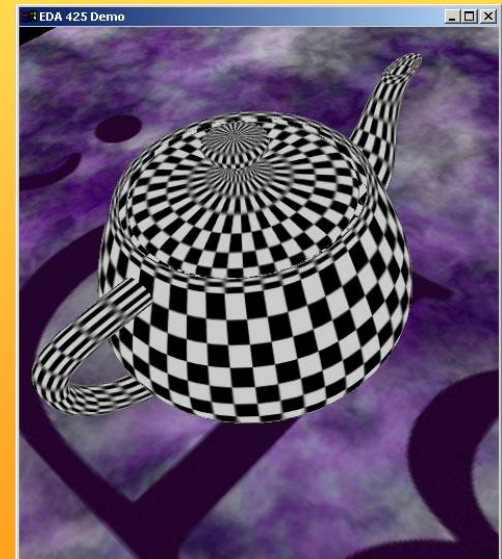
- Purpose: more realism, and this is a cheap way to do it



+

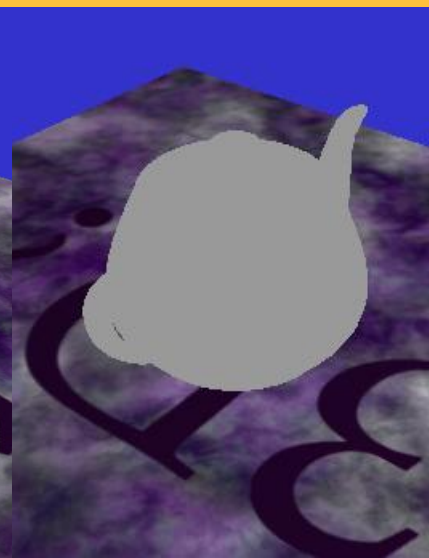
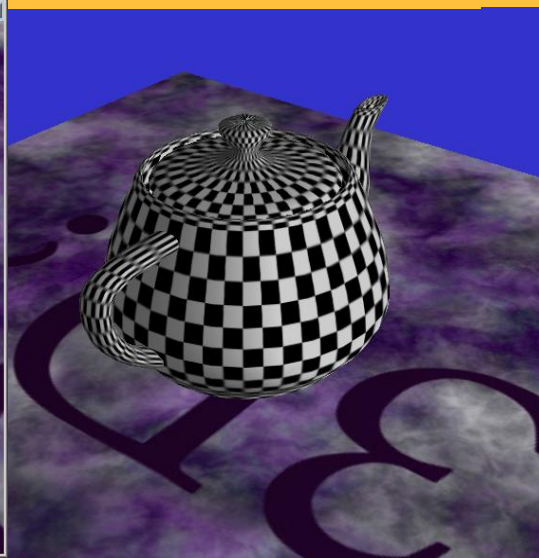
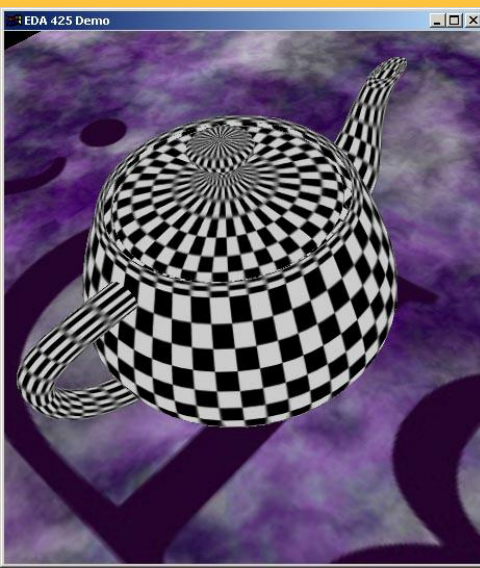
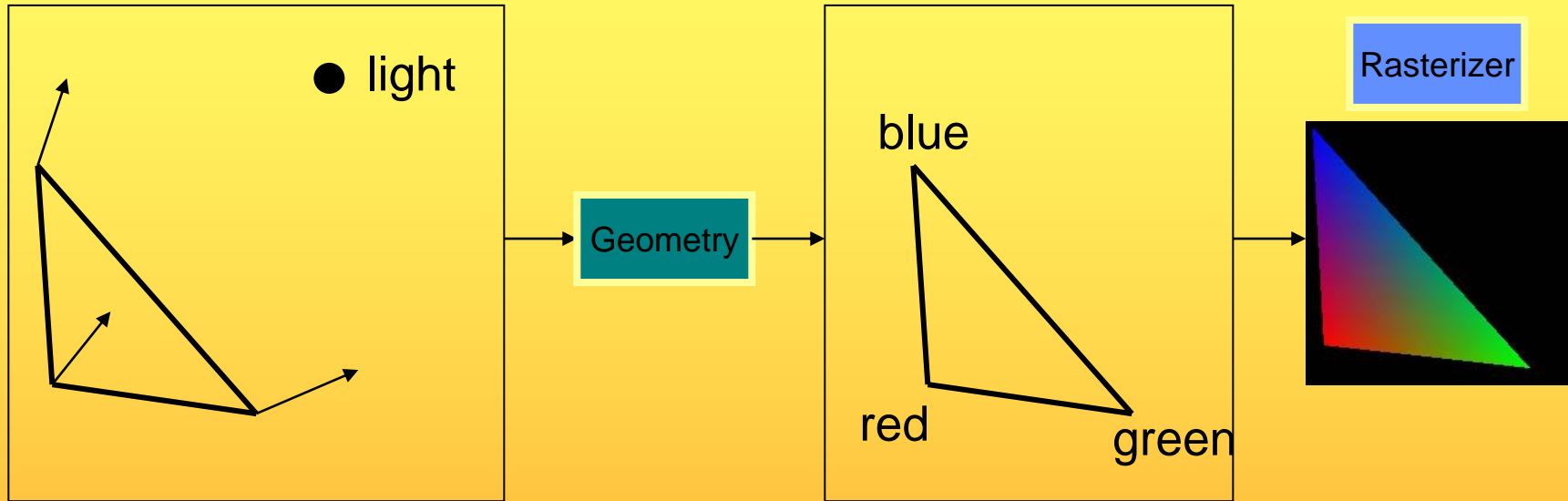


=





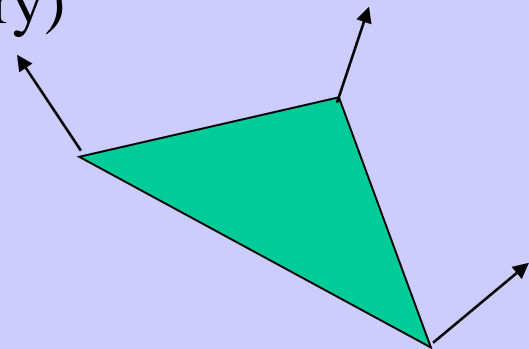
# Lighting computation per triangle vertex



# The Graphics Rendering Pipeline

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

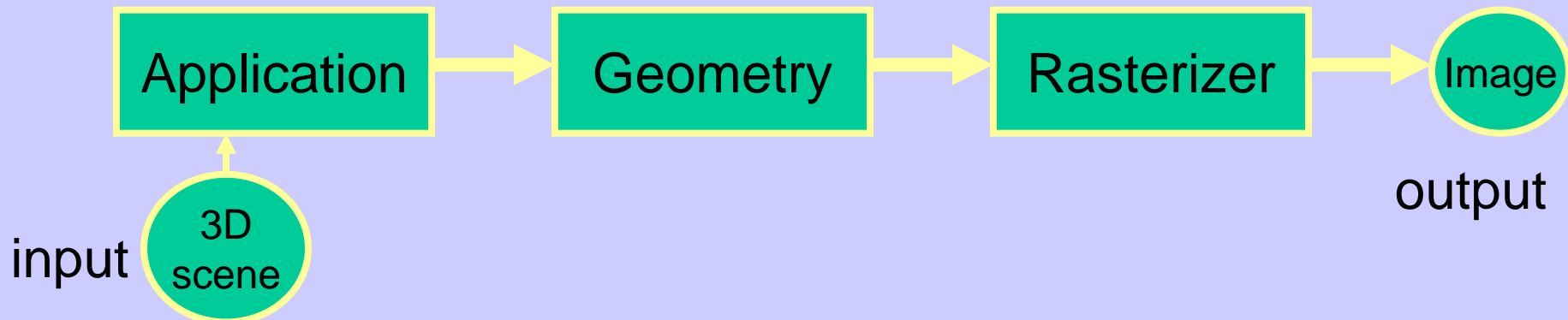
- First, of all to take a picture, it takes a camera – a virtual one.
  - Decides what should end up in the final image
- A 3D scene is:
  - Geometry (triangles, lines, points, and more)
  - Light sources
  - Material properties of geometry
  - Textures (images to glue onto the geometry)
- A triangle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



# 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

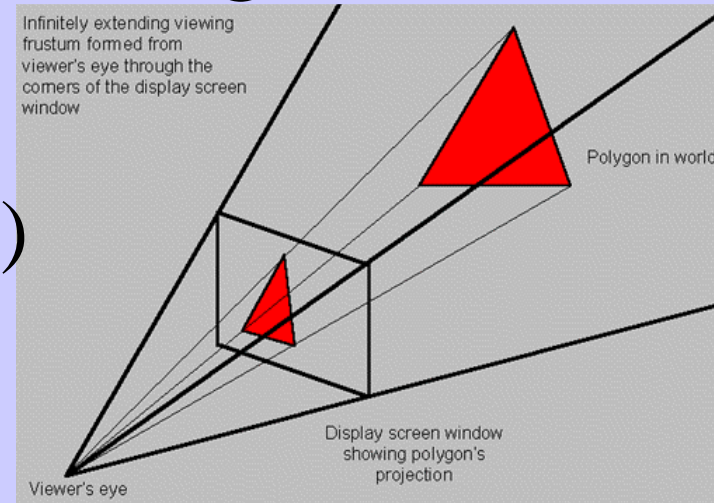


# The APPLICATION stage

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

# The GEOMETRY stage

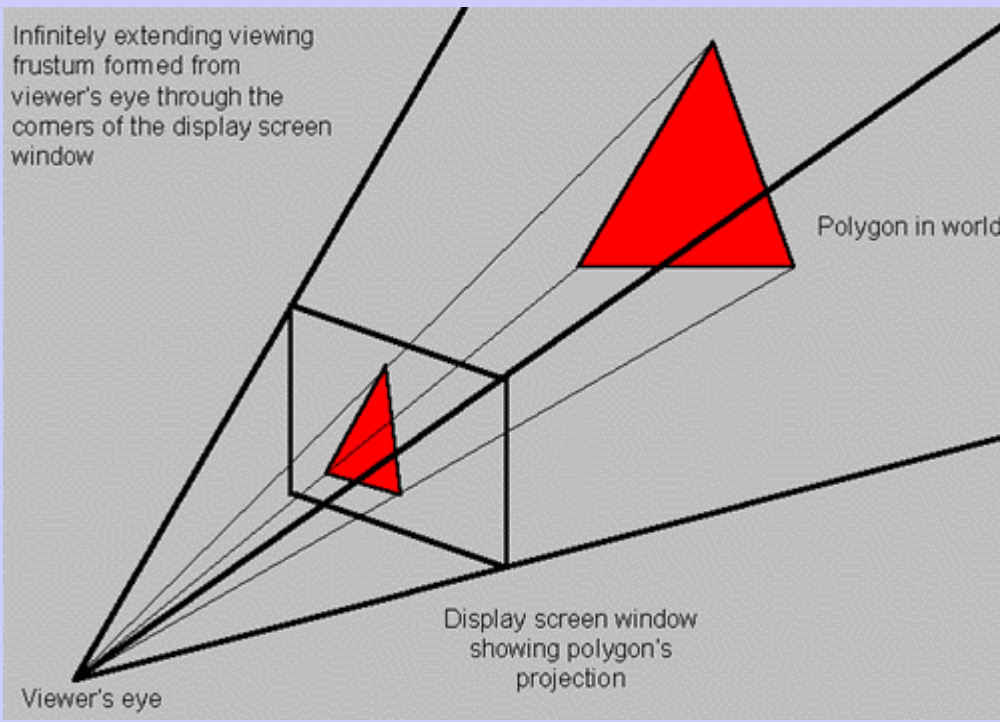
- Task: "geometrical" operations on the input data (e.g. triangles)
- Allows:
  - Move objects (matrix multiplication)
  - Move the camera (matrix multiplication)
  - Lighting computations per triangle vertex
  - Project onto screen (3D to 2D)
  - Clipping (avoid triangles outside screen)
  - Map to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



# The GEOMETRY stage

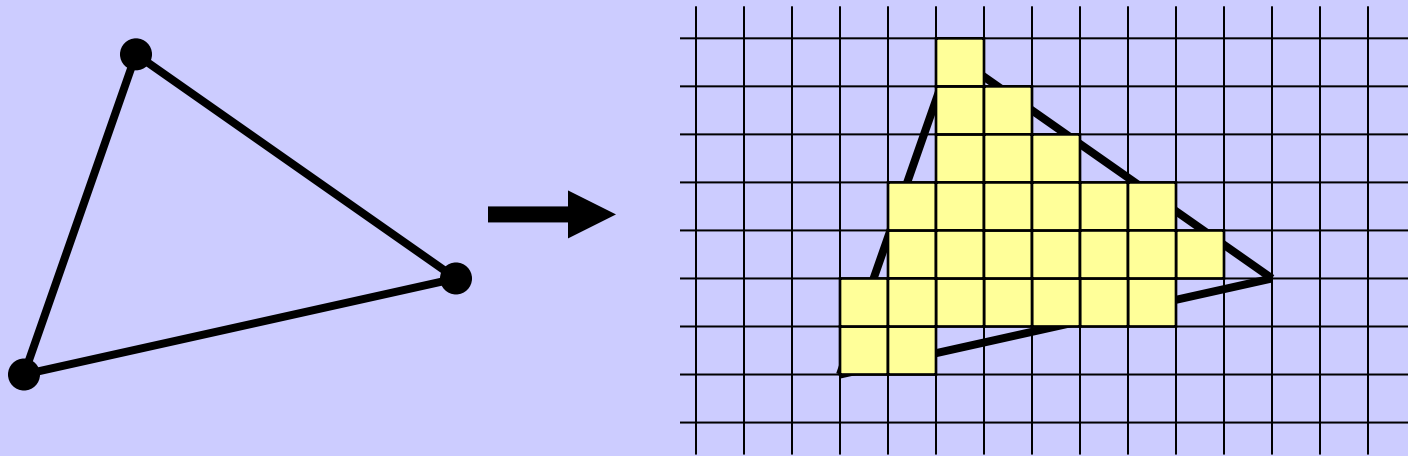


- (Instances)
- Vertex Shader
  - A program executed per vertex
    - Transformations
    - Projection
- Clipping
- Screen Mapping



# The RASTERIZER stage

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



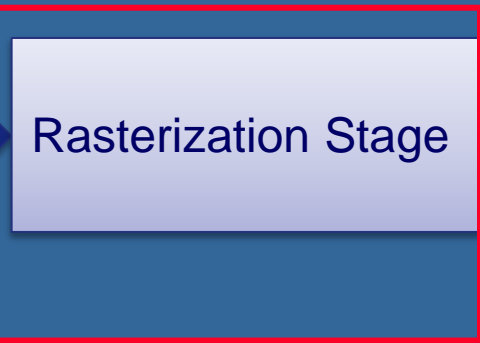
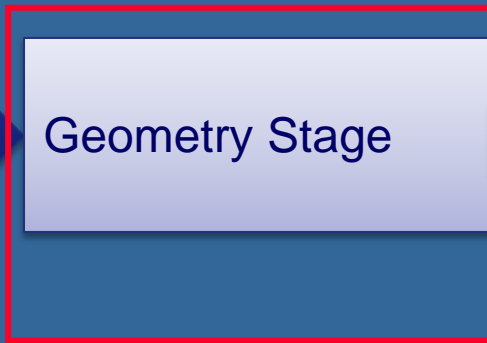
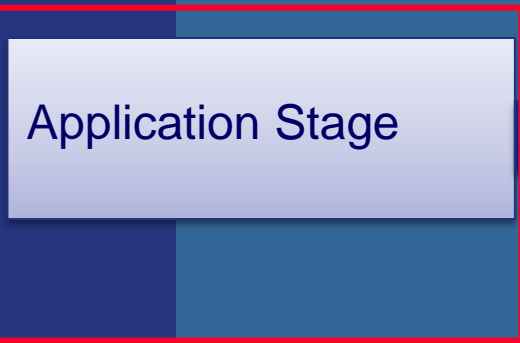
- Computes color per pixel, using fragment shader (=pixel shader)
  - textures and various other per-pixel operations
- And visibility is resolved here: sorts the primitives in the z-direction



# Rendering Pipeline and Hardware

CPU

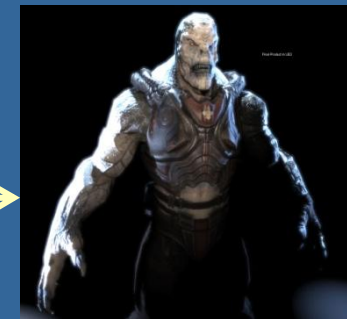
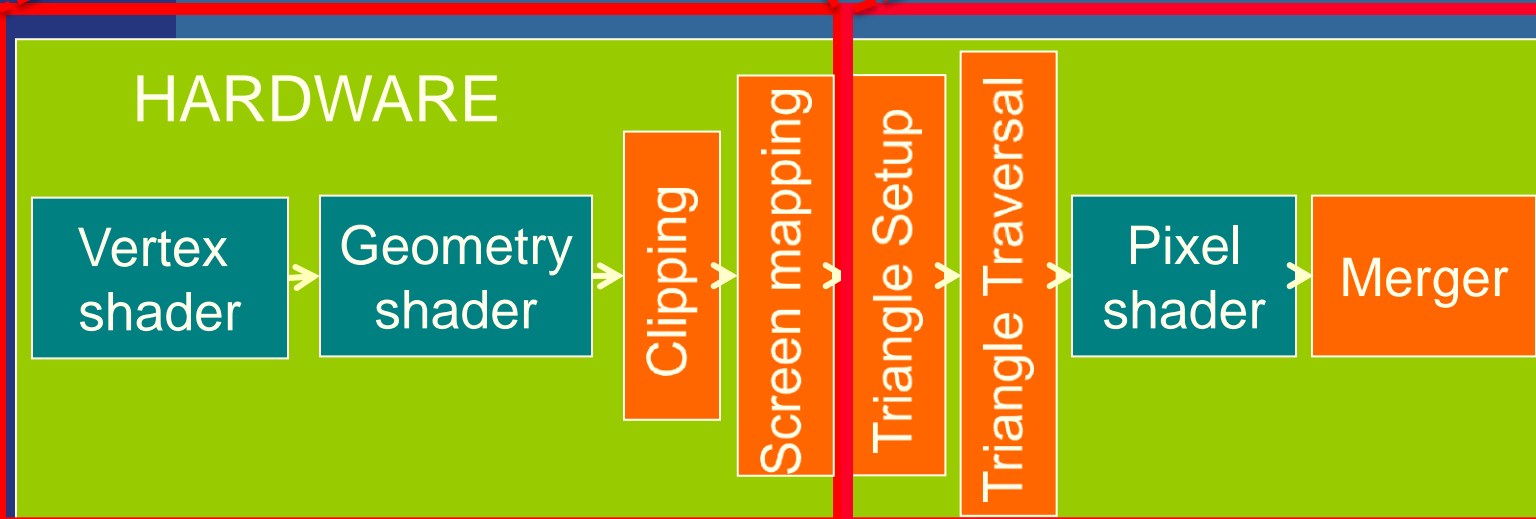
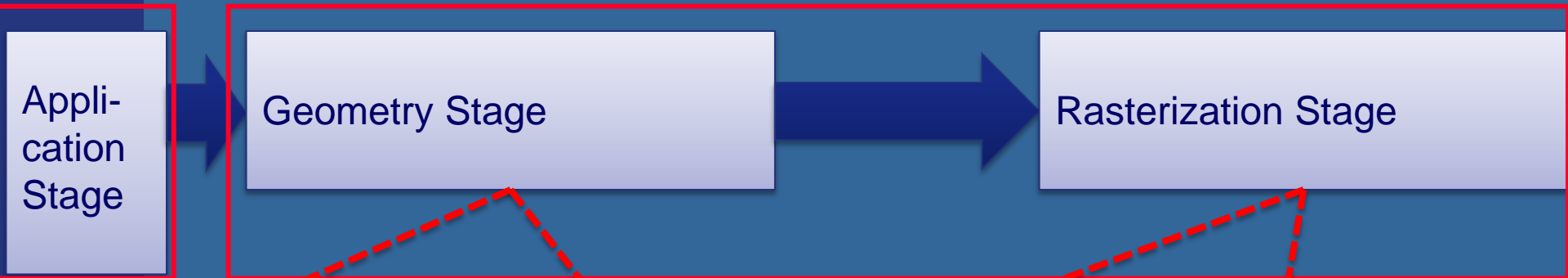
GPU



# Rendering Pipeline and Hardware

CPU

GPU

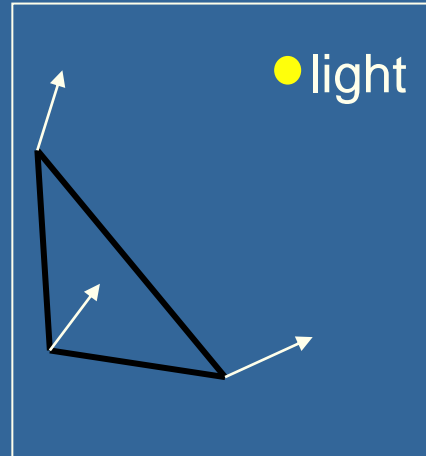
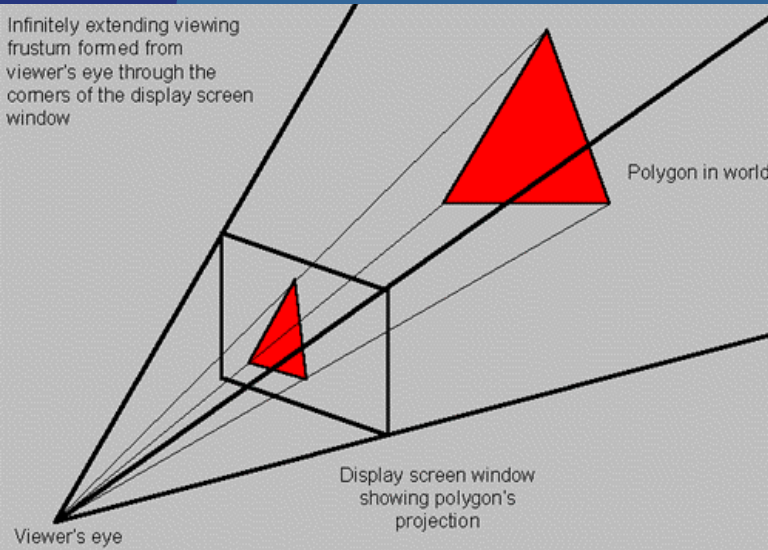


Display

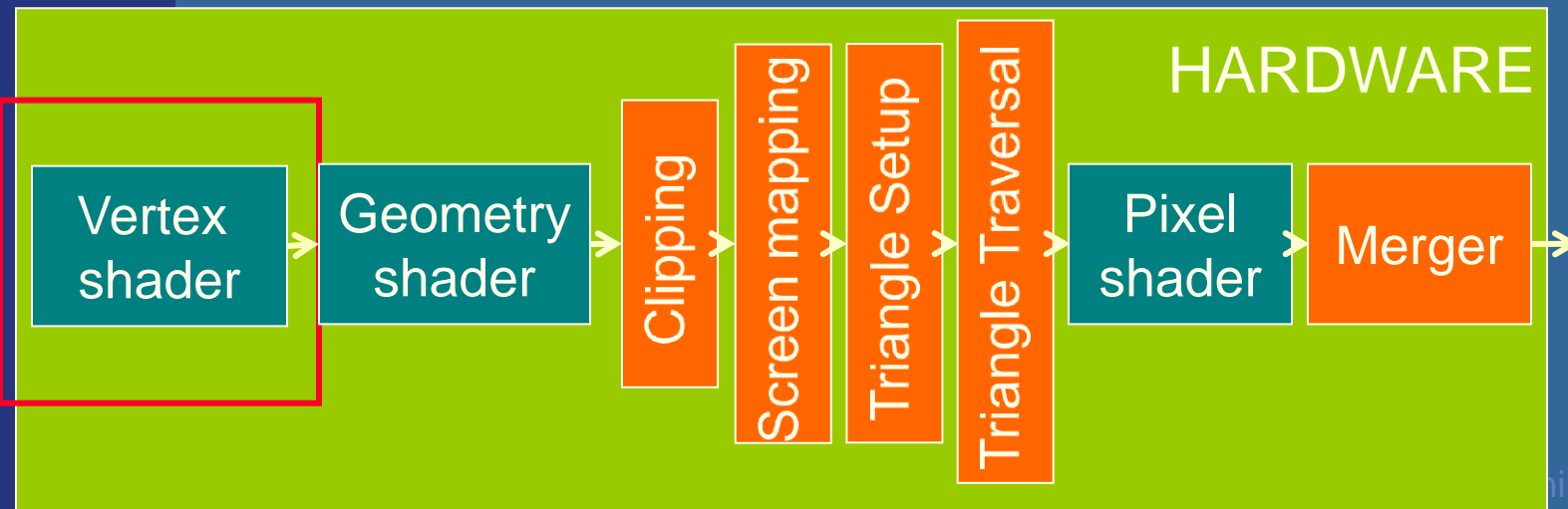
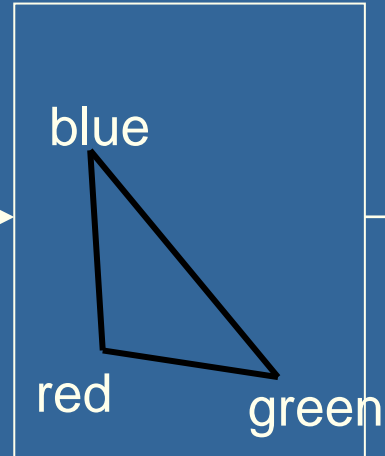
# Hardware design

Vertex shader:

- Lighting (colors)
- Screen space positions



Geometry

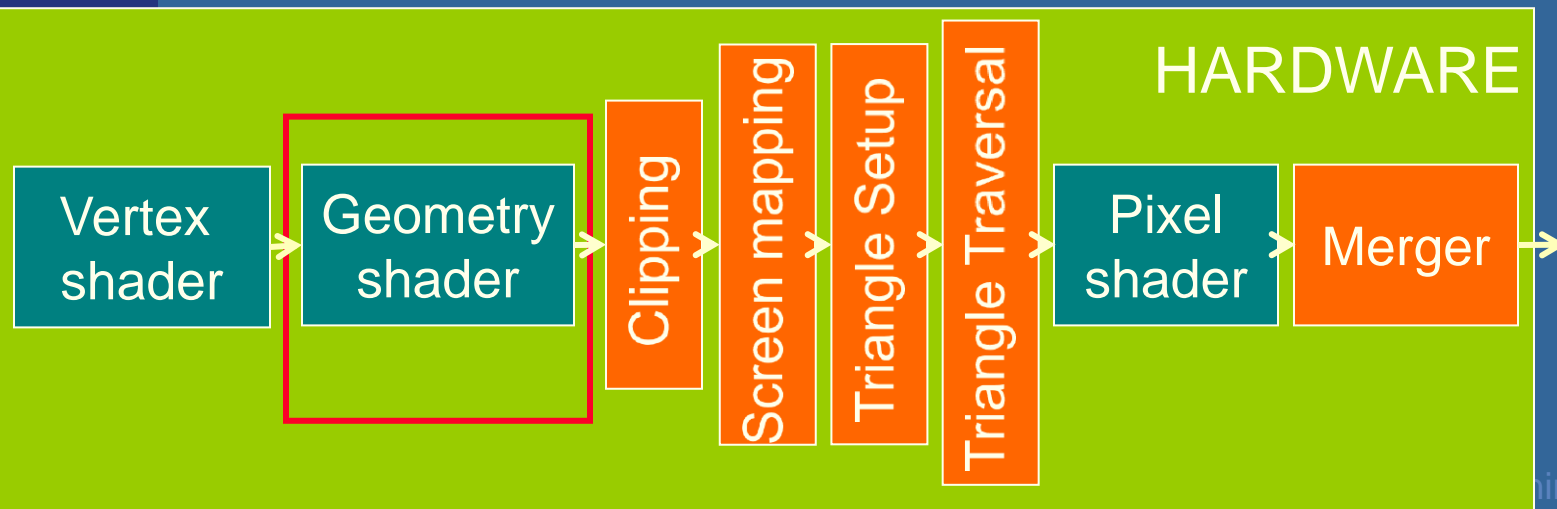
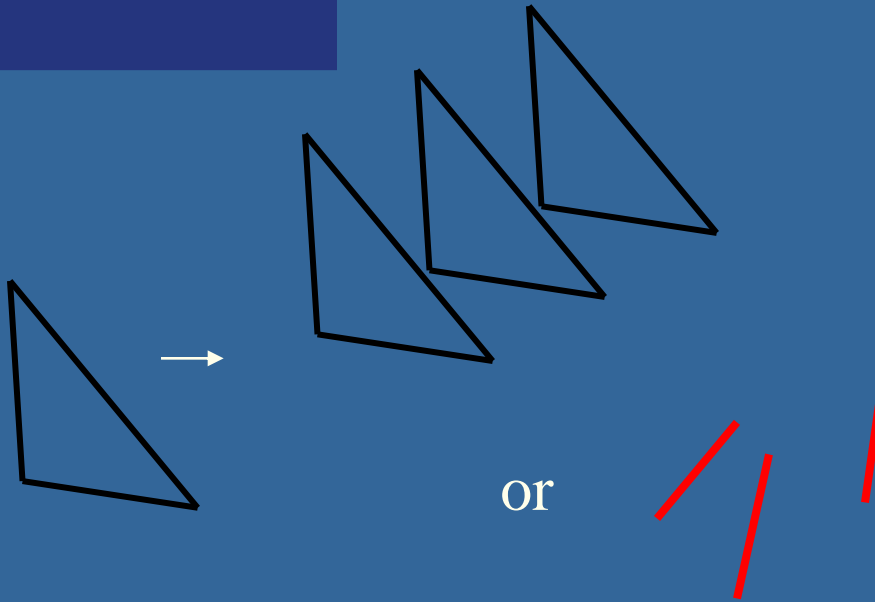


Display

# Hardware design

Geometry shader:

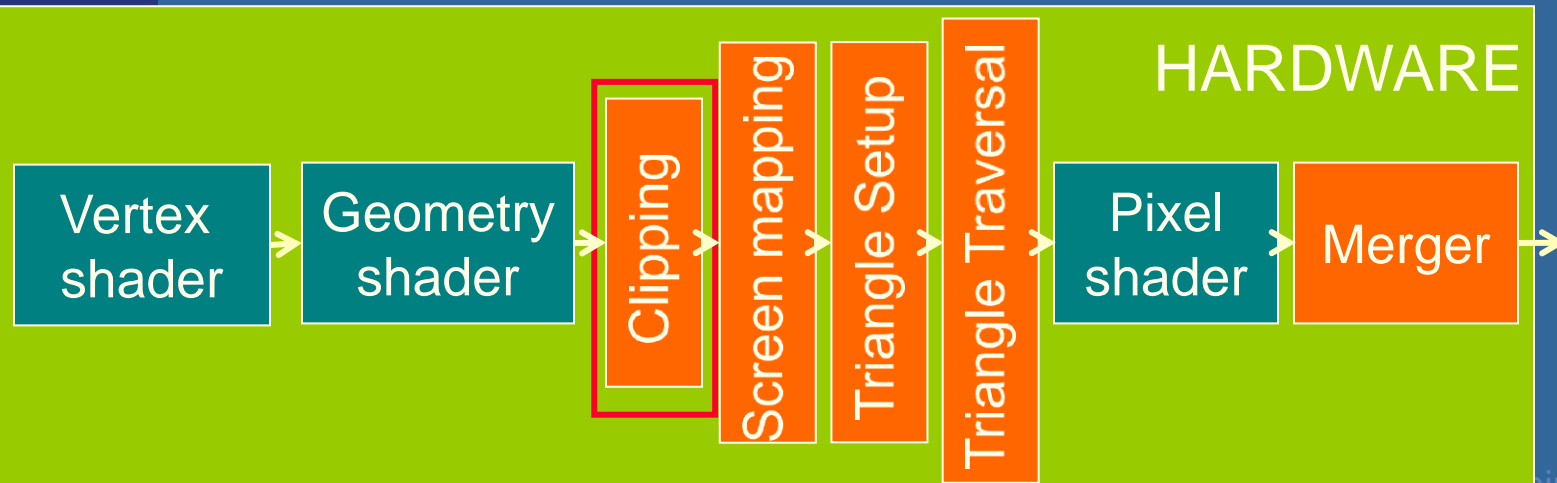
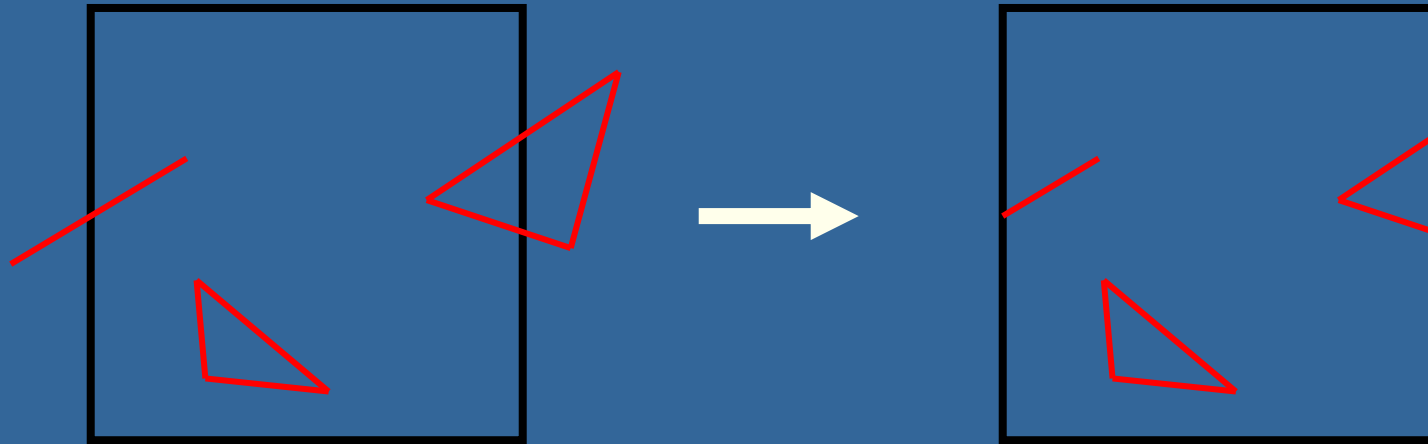
- One input primitive
- Many output primitives



Display

# Hardware design

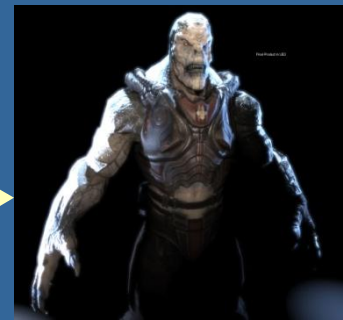
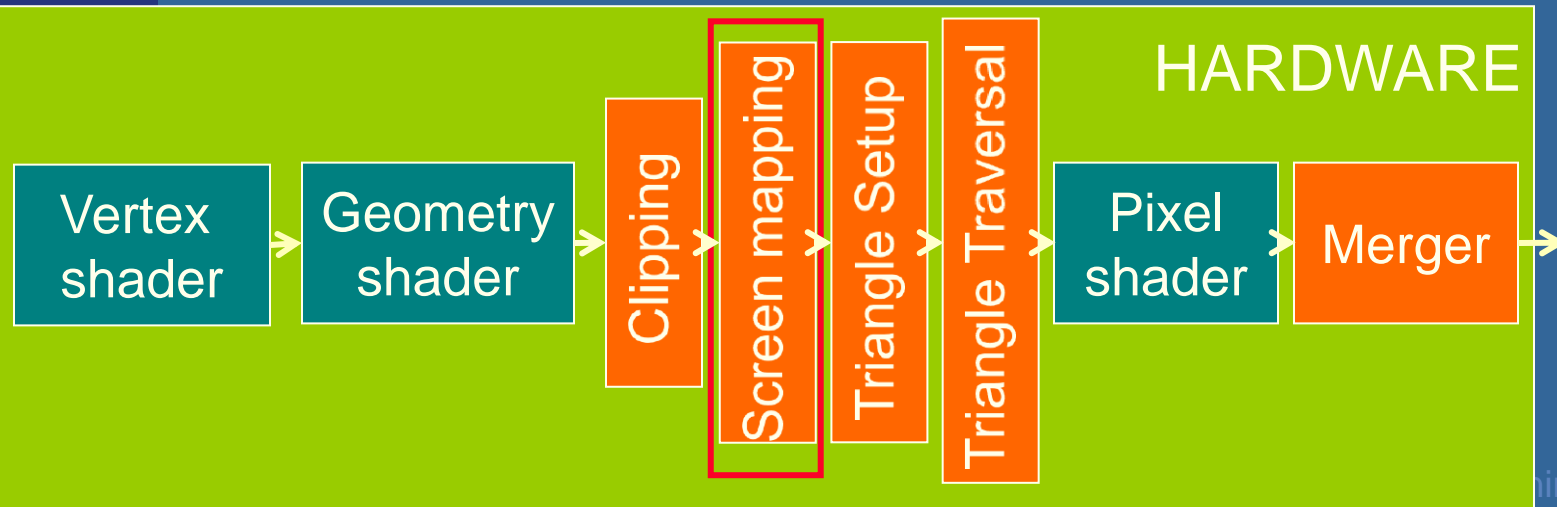
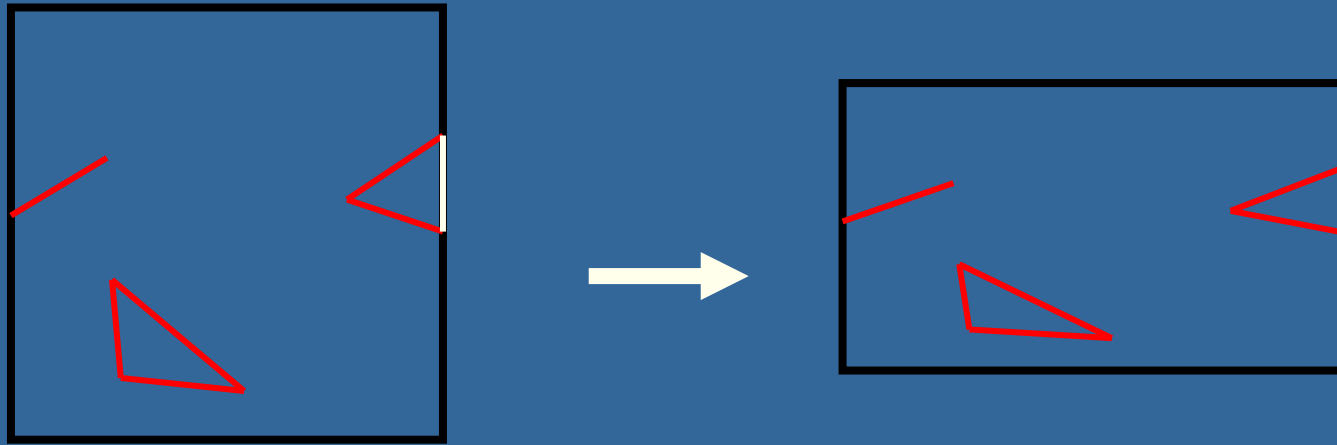
Clips triangles against the unit cube (i.e., "screen borders")



Display

# Hardware design

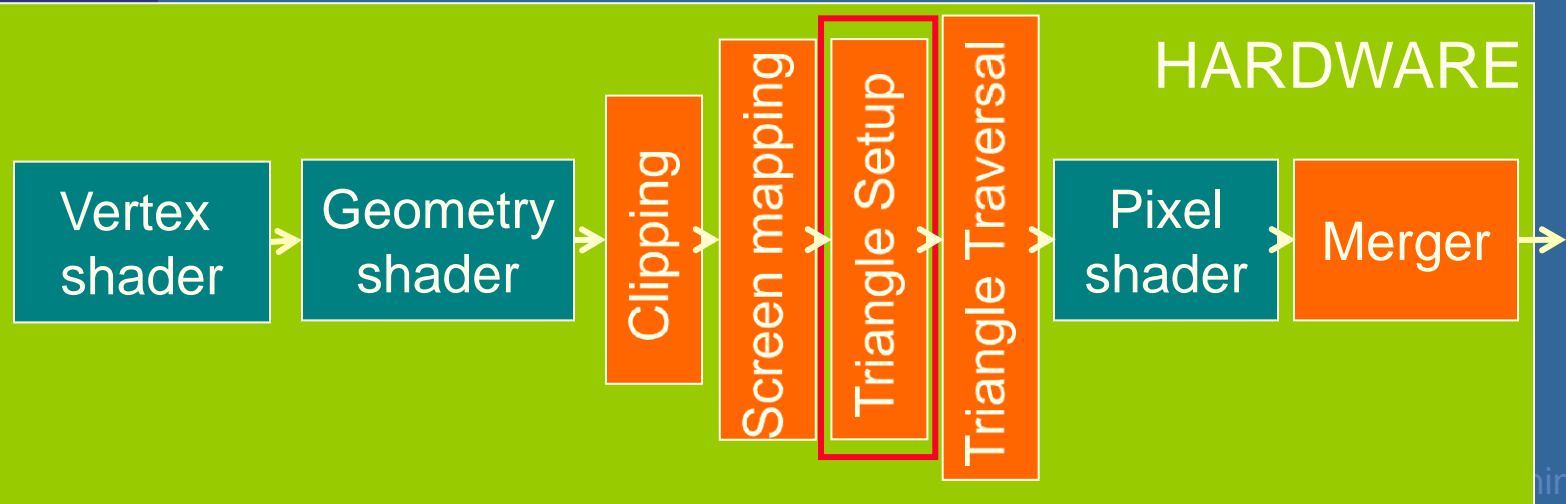
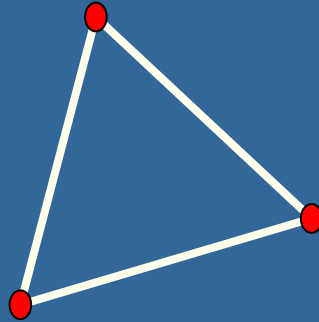
Maps window size to unit cube



Display

# Hardware design

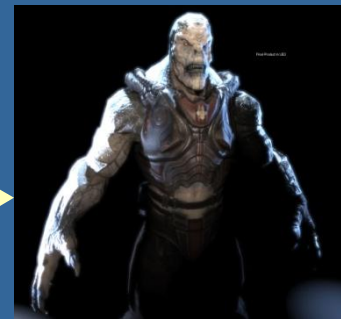
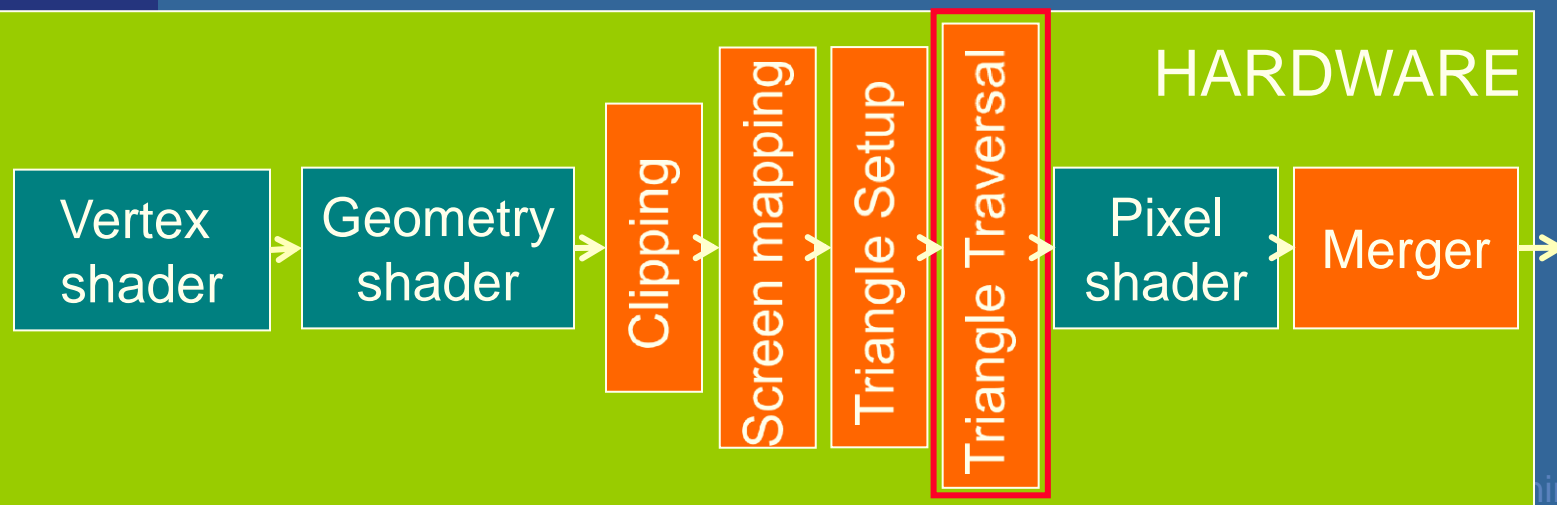
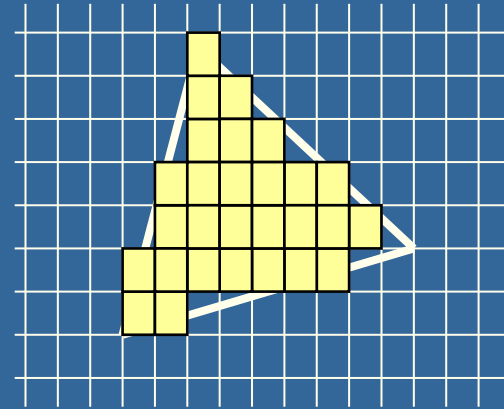
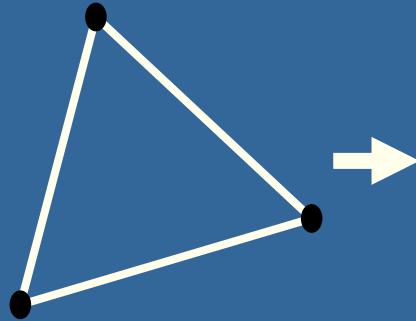
Collects three vertices into one triangle



Display

# Hardware design

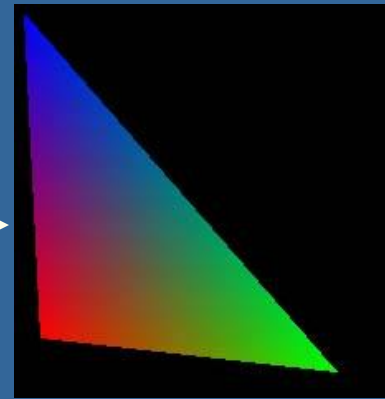
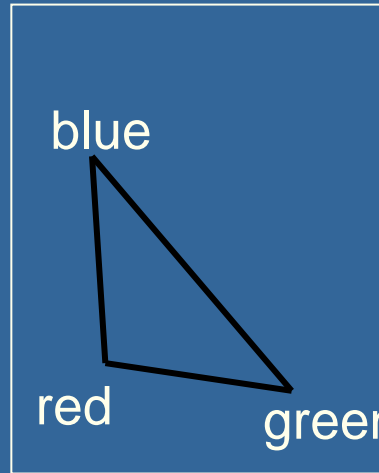
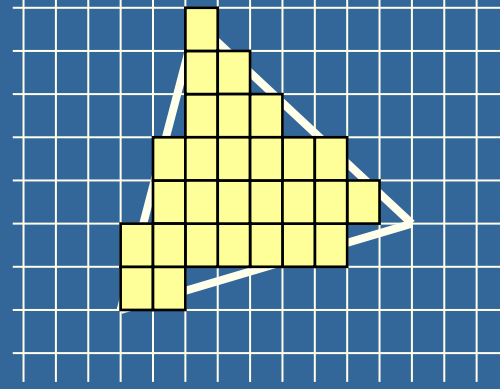
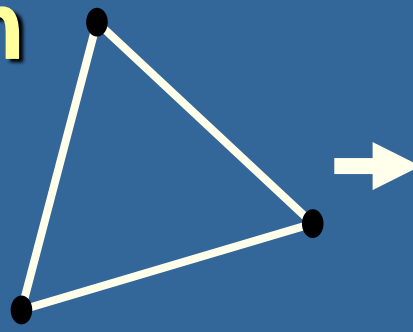
Creates the fragments/pixels for the triangle



Display



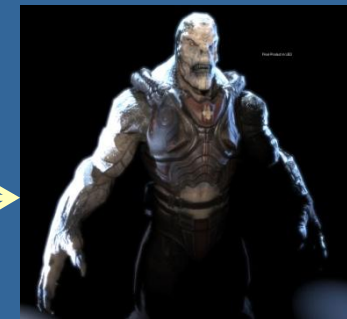
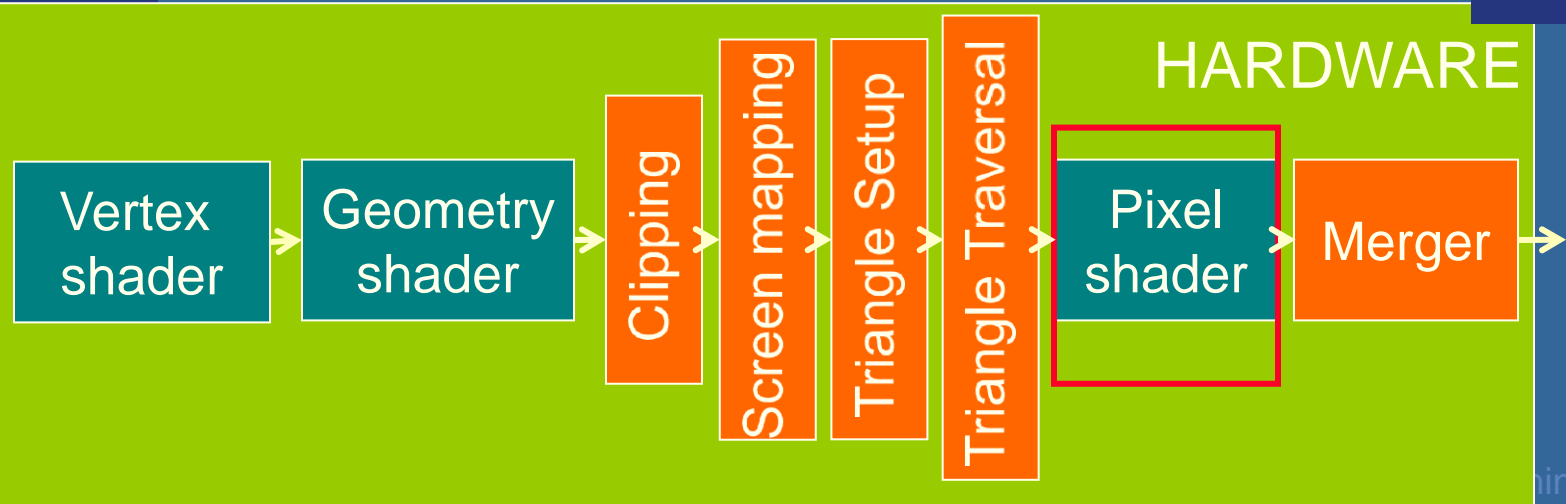
# Hardware design



Rasterizer

Pixel Shader:  
Compute color using:

- Textures
- Interpolated data (e.g. Colors + normals) from vertex shader



Display

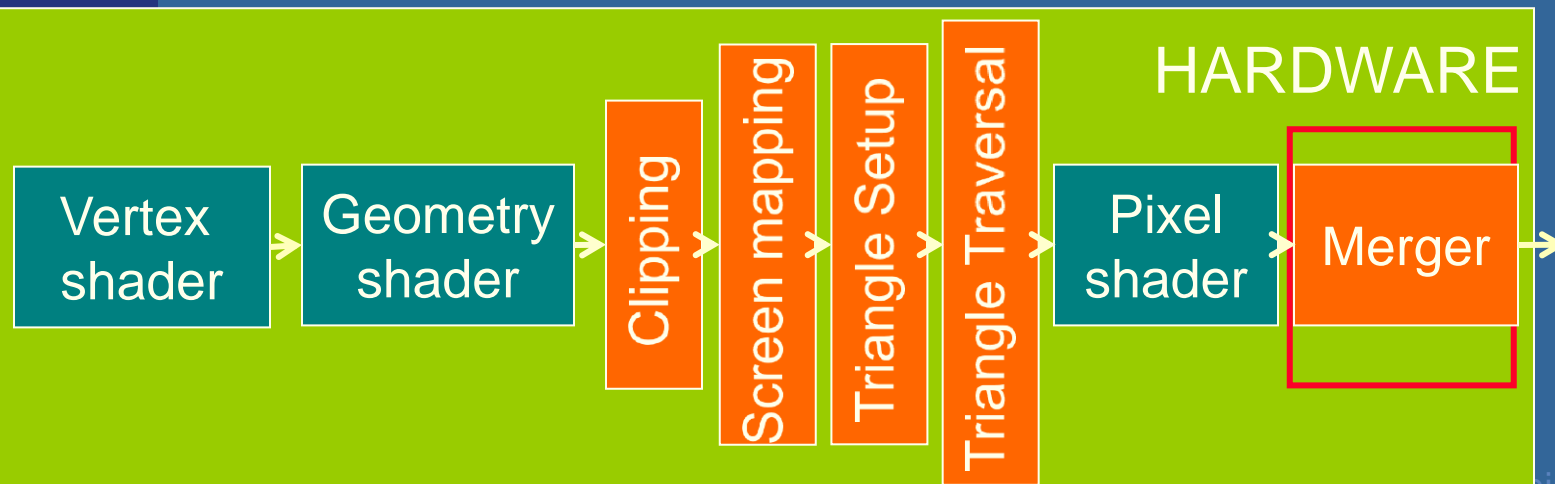
# Hardware design

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



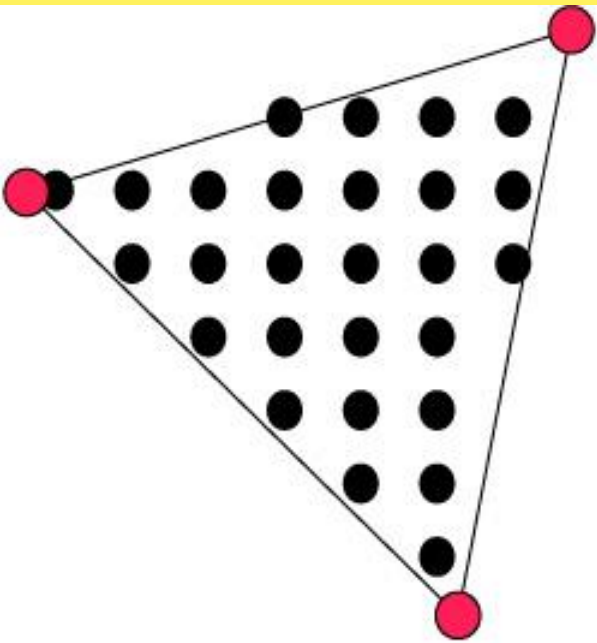
Frame buffer:

- Color buffers
- Depth buffer
- Stencil buffer



Display

# What is vertex and fragment (pixel) shaders?



- **Memory:** Texture memory (read + write) typically 500 Mb – 4 GB
- **Program size:** the smaller the faster
- **Instructions:** mul, rcp, mov, dp, rsq, exp, log, cmp, jnz...

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

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

# Cg - "C for Graphics" (NVIDIA)

```
if (slice >= 0.0h) {
    half gradedEta = BallData.ETA;
    gradedEta = 1.0h/gradedEta; // test hack
    half3 faceColor = BgColor; // blown out - go to BG color
    half c1 = dot(-Vn,Nf);
    half cs2 = 1.0h-gradedEta*gradedEta*(1.0h-c1*c1);

    if (cs2 >= 0.0h) {
        half3 refVector = gradedEta*Vn+((gradedEta*c1+sqrt(cs2))*Nf);
        // now let's intersect with the iris plane
        half irist = intersect_plane(IN.OPosition,refVector,planeEquation);
        half fadeT = irist * BallData.LENS_DENSITY;
        fadeT = fadeT * fadeT;
        faceColor = DiffPupil.xxx; // temporary (?)
        if (irist > 0) {
            half3 irisPoint = IN.OPosition + irist*refVector;
            half3 irisST = (irisScale*irisPoint) + half3(0.0h,0.5h,0.5h);
            faceColor = tex2D(ColorMap,irisST.yz);
        }
        faceColor = lerp(faceColor,LensColor,fadeT);
        hitColor = lerp(missColor,faceColor,smoothstep(0.0h,GRADE,slice));
    }
}
```

# PixelShader 3.0

```

// if (-dir.z/|dir| > cos(PI/4)) t1 = zero
dp3 r6.w, r6, r6
rsq r6.w, r6.w ← normalization
mad r0.w, -r6.z, r6.w, -CosPiOverFour
cmp r10.y, r0.w, Zero, r10.y

// set r10 to 0 if Disc <= 0
cmp r10.xy, -r7.w, Zero, r10

// compute r1 and r2 clipped
mad r1.xyz, r6, r10.x, r4 // IPO
mad r2.xyz, r6, r10.y, r4 // IP1
|
// project
rcp r11.w, r1.z
mad r1.xyz, r1, r11.w, NegZ // P0
rcp r11.w, r2.z
mad r2.xyz, r2, r11.w, NegZ // P1

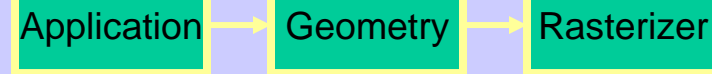
// Compute area
texld r3, r1, ATan2Texture // theta0
texld r4, r2, ATan2Texture // theta1

crs r5.z, r1, r2 // z = 2
abs r5.z, r5.z

mov r3.y, r4.x
texld r4, r3, SphAreaTexture // lookup theta/PI

```

- Float, int
- Instructions operate on 1,2,3 or 4 components
  - x,y,z,w or
  - r,g,b,a
- Free Swizzling
- Only read from texture
- (Only write to pixel (8 output buffers))



# Rewind!

## Let's take a closer look

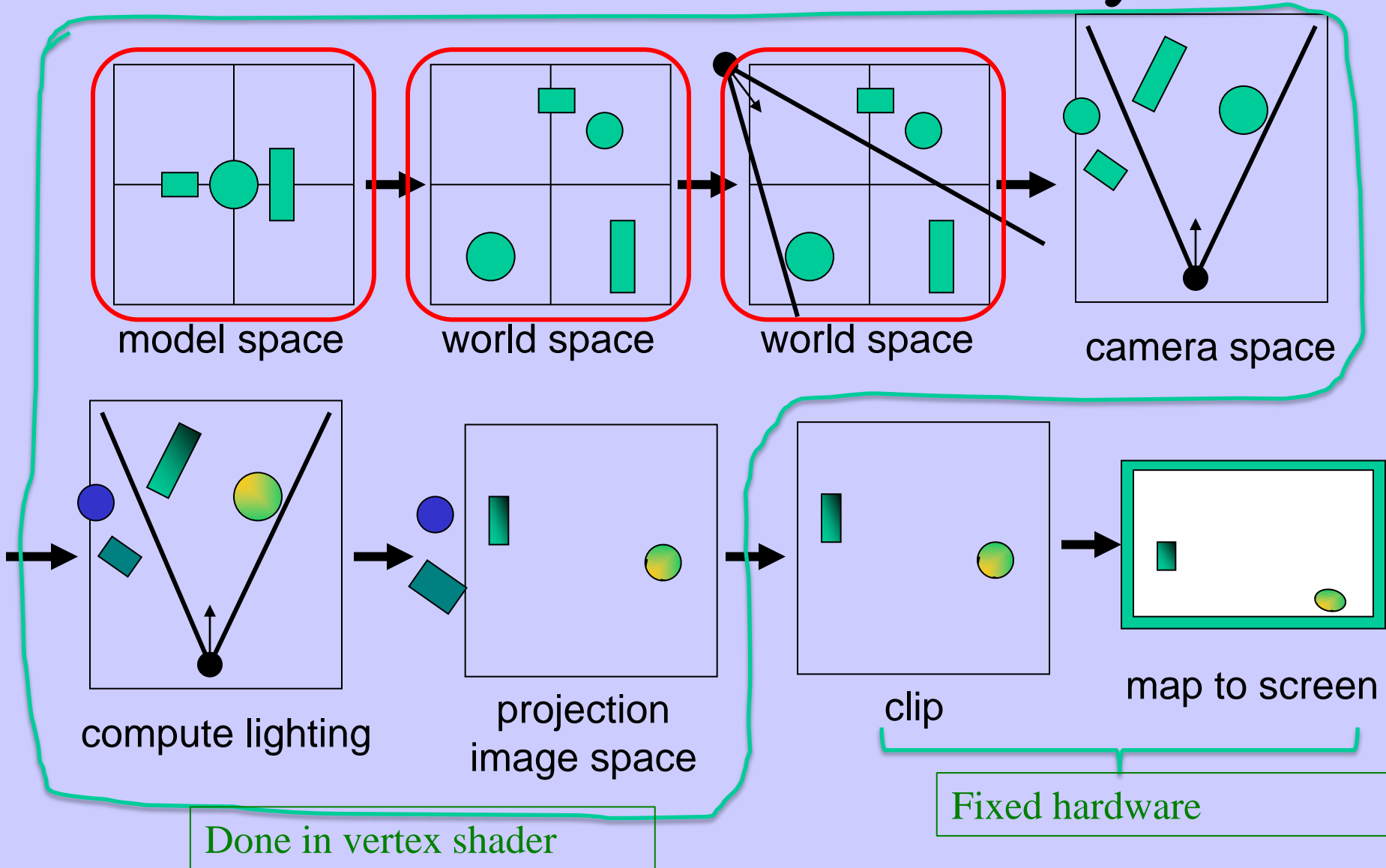
- The programmer "sends" down primitives to be rendered through the pipeline (using API calls)
- The geometry stage does per-vertex operations (and per-triangle operations)
- The rasterizer stage does per-pixel operations
- Next, scrutinize geometry and rasterizer

Application

Geometry

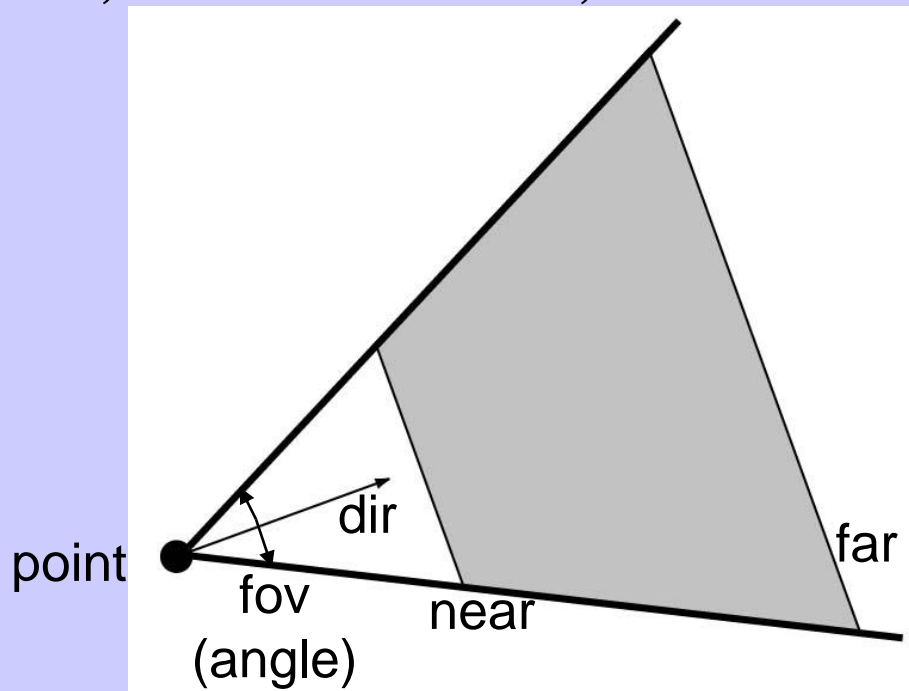
Rasterizer

# GEOMETRY - Summary



# Virtual Camera

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

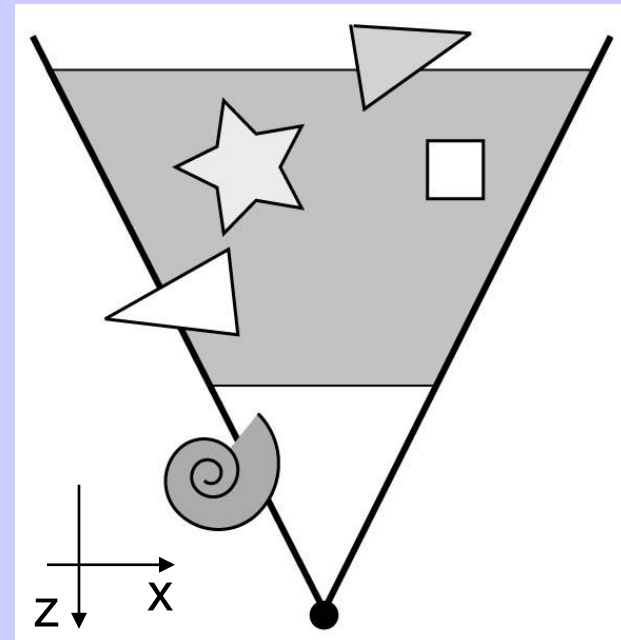
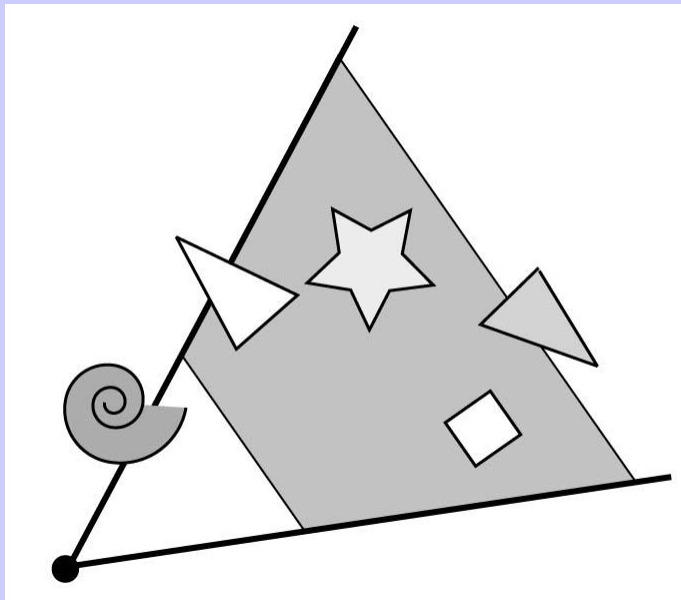


- Create image of geometry inside gray region
- Used by OpenGL, DirectX, ray tracing, etc.



# 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

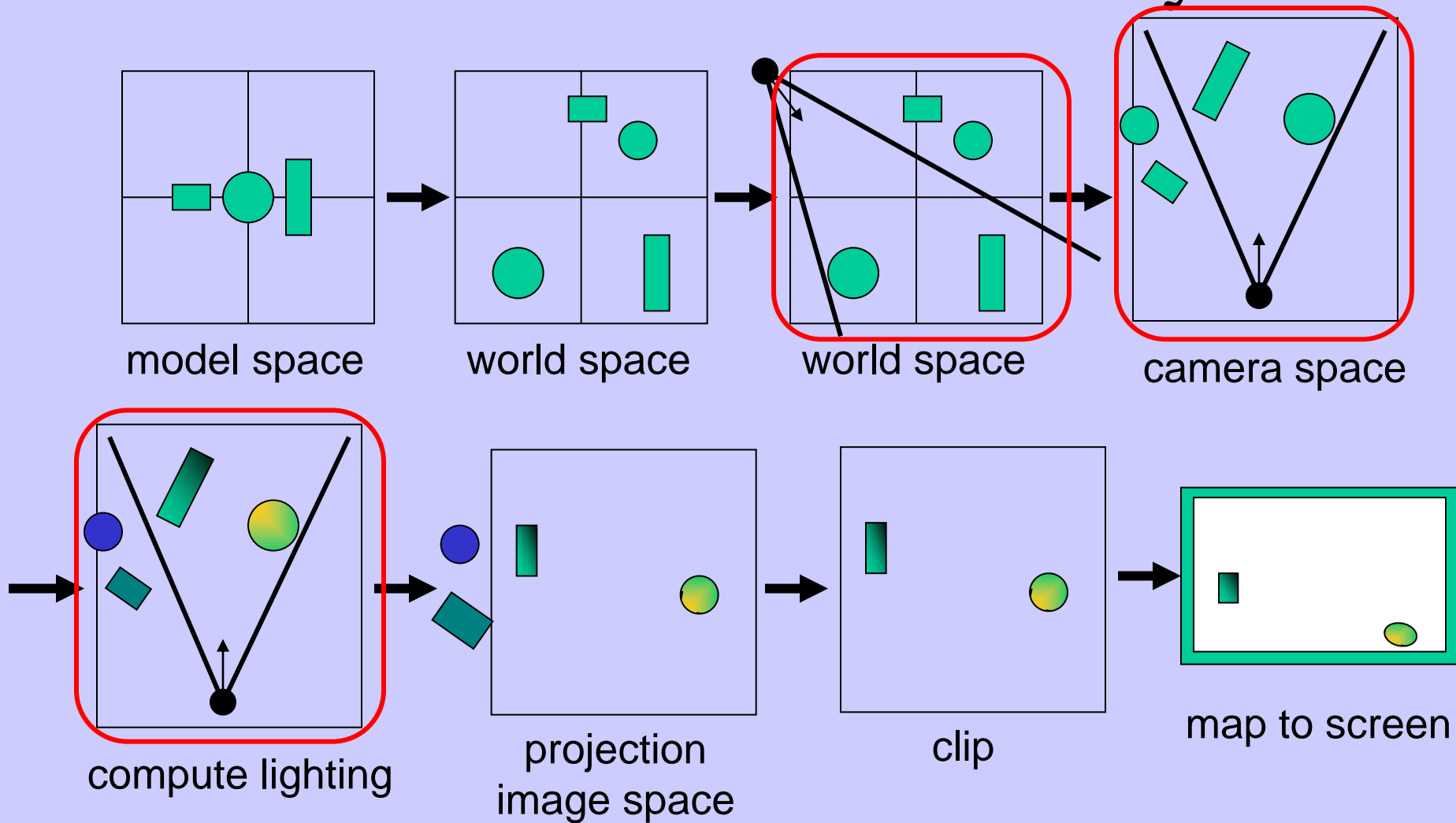


Application

Geometry

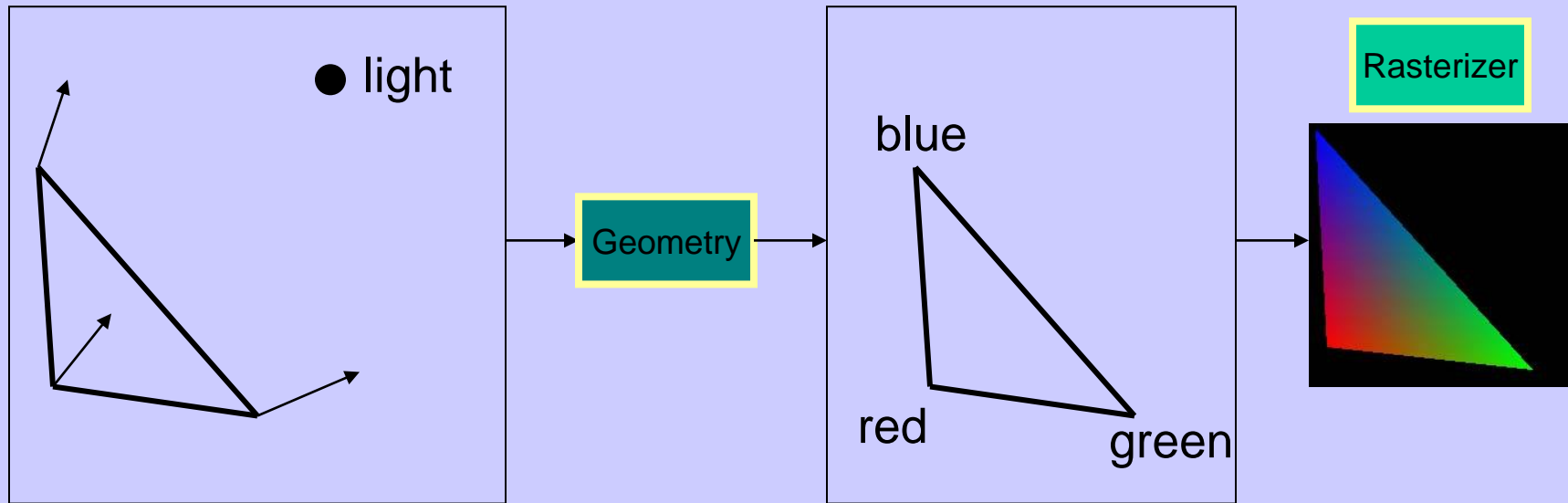
Rasterizer

# GEOMETRY - Summary



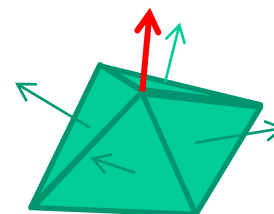
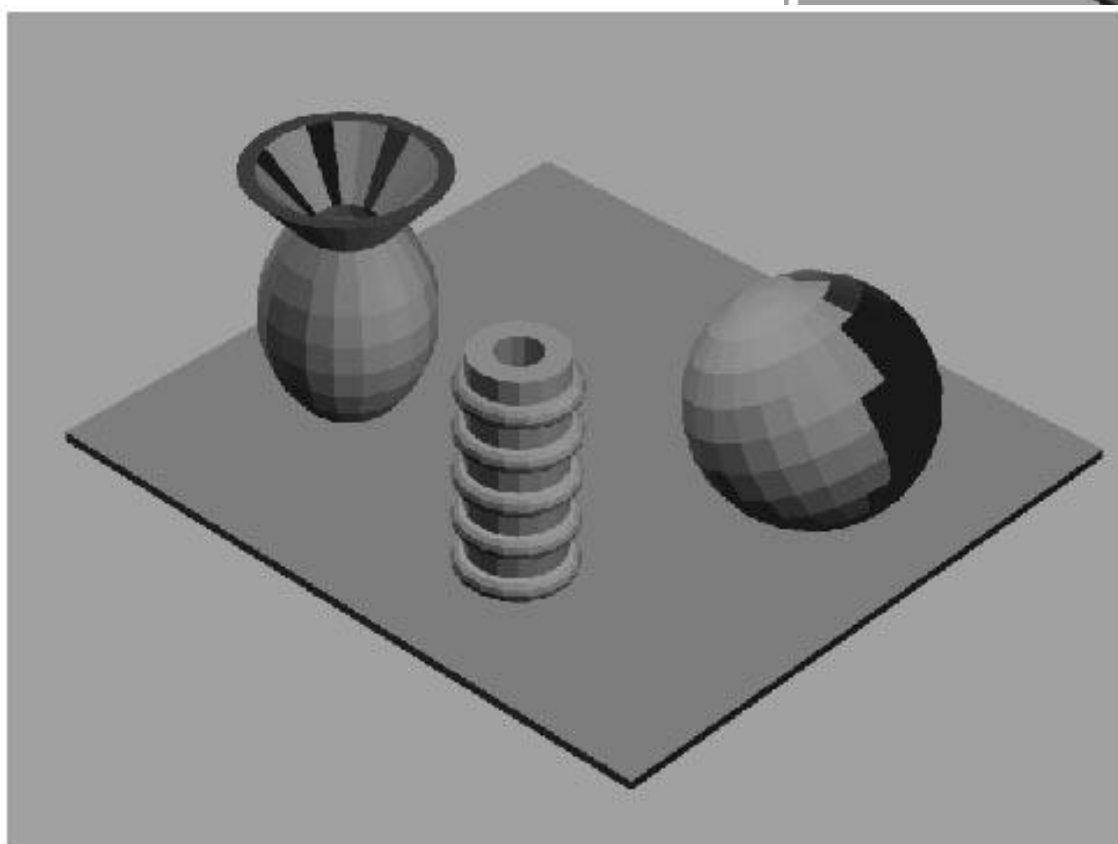
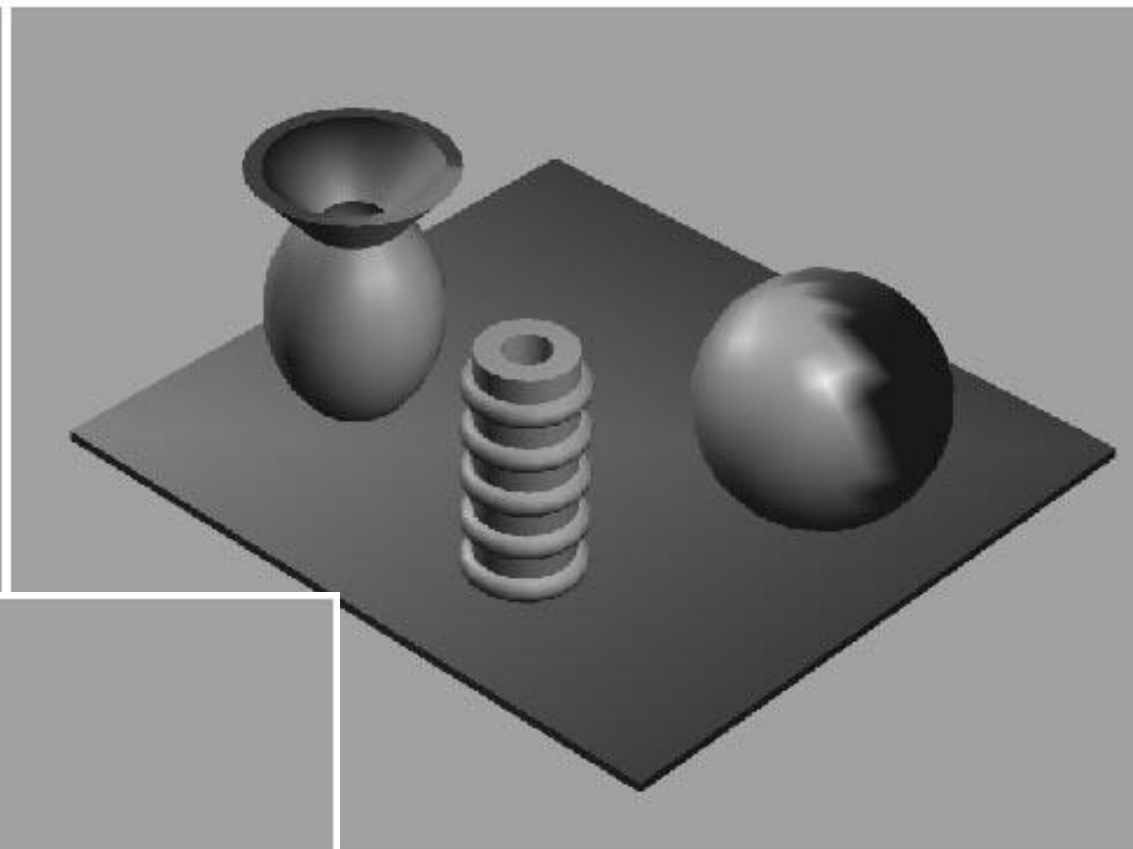
# GEOMETRY - Lighting

- 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

# Example

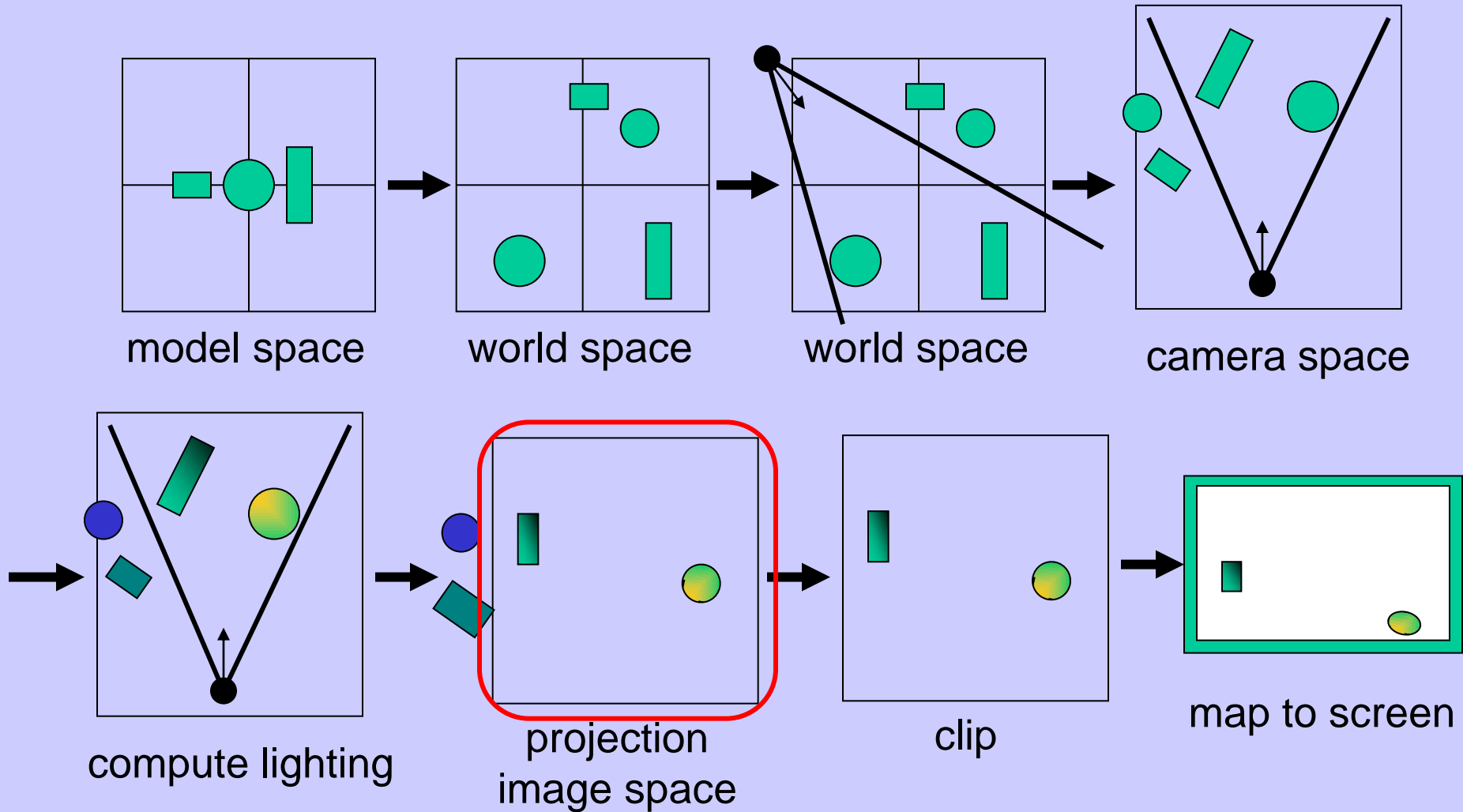


Application

Geometry

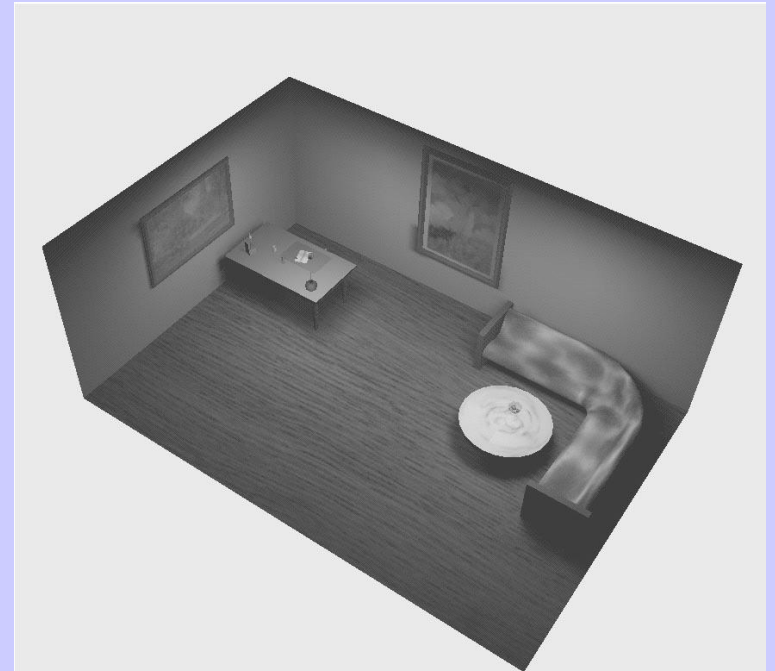
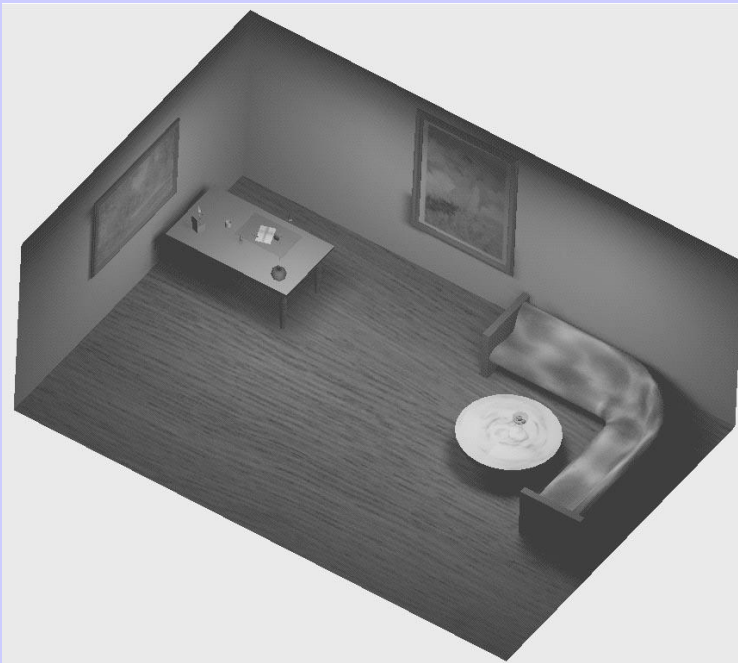
Rasterizer

# GEOMETRY - Summary



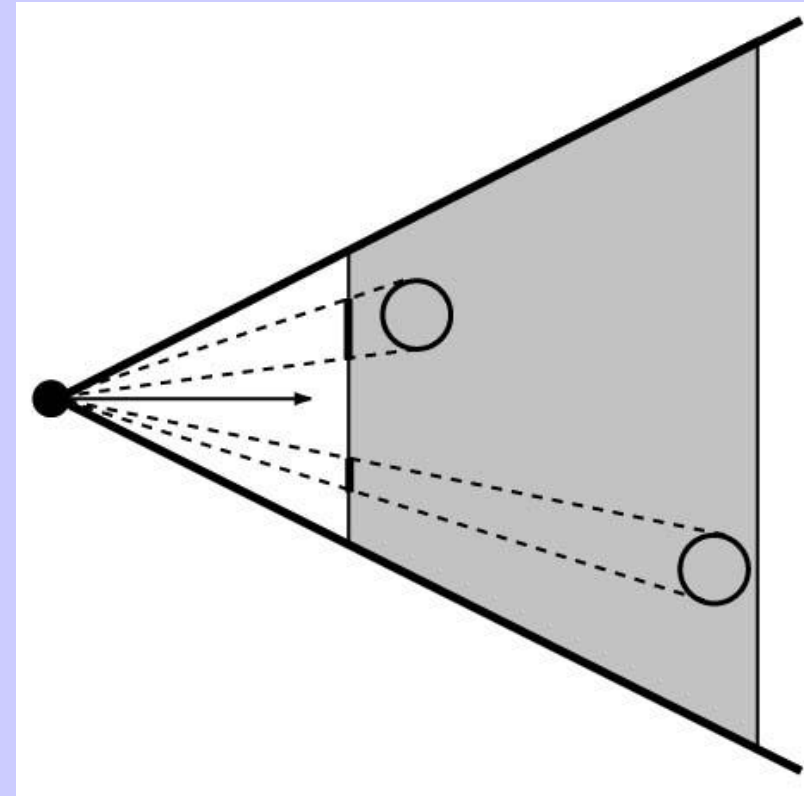
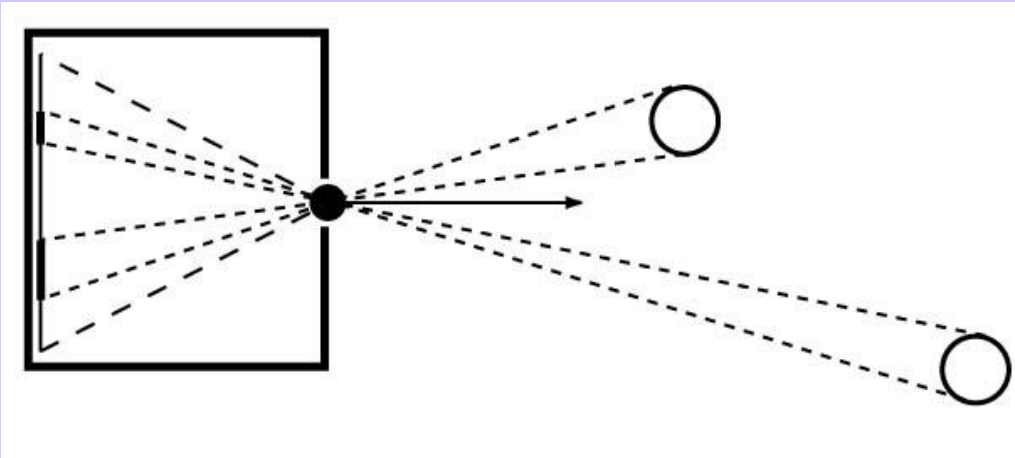
# GEOMETRY - Projection

- 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)

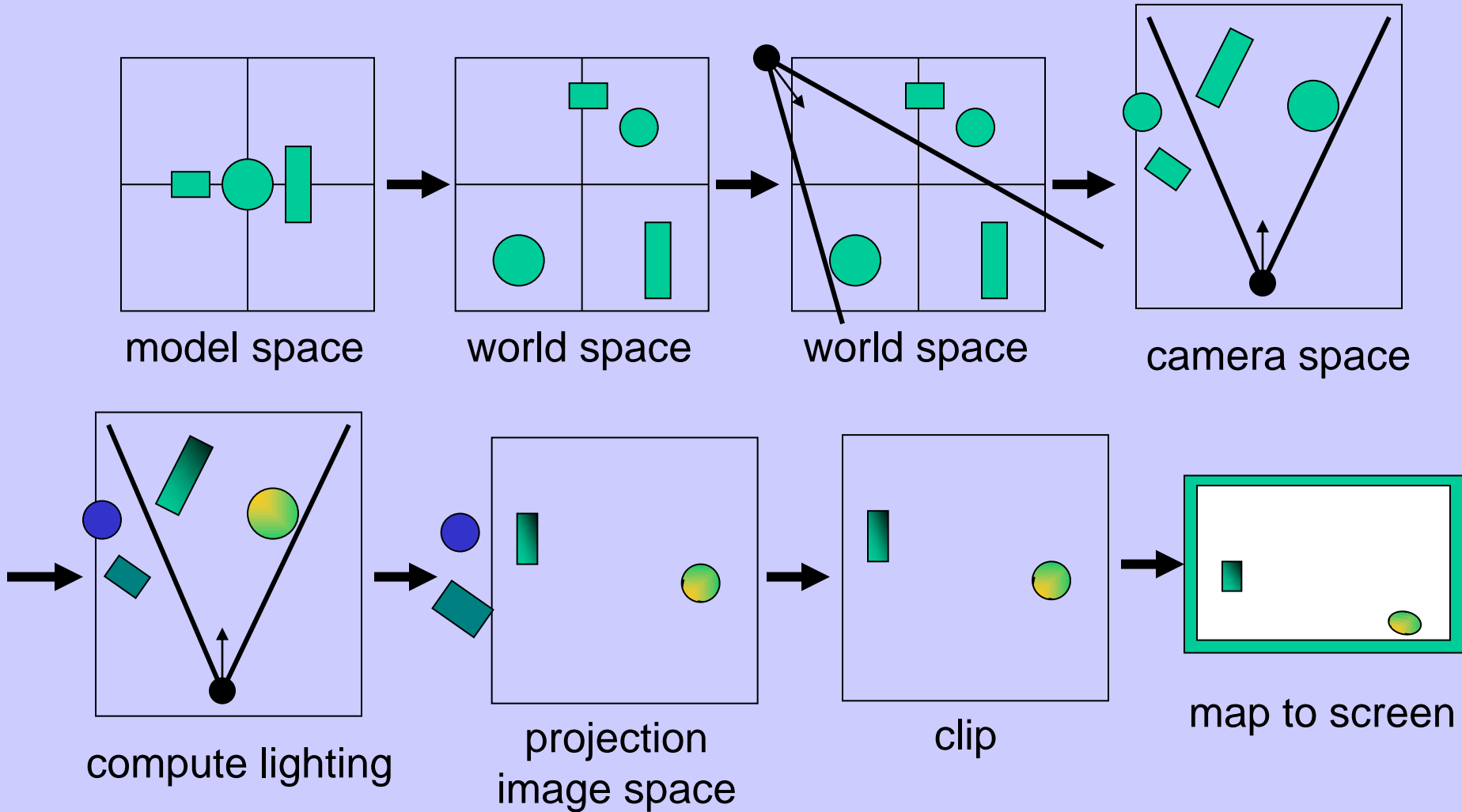


Application

Geometry

Rasterizer

# GEOMETRY - Summary





# GEOMETRY

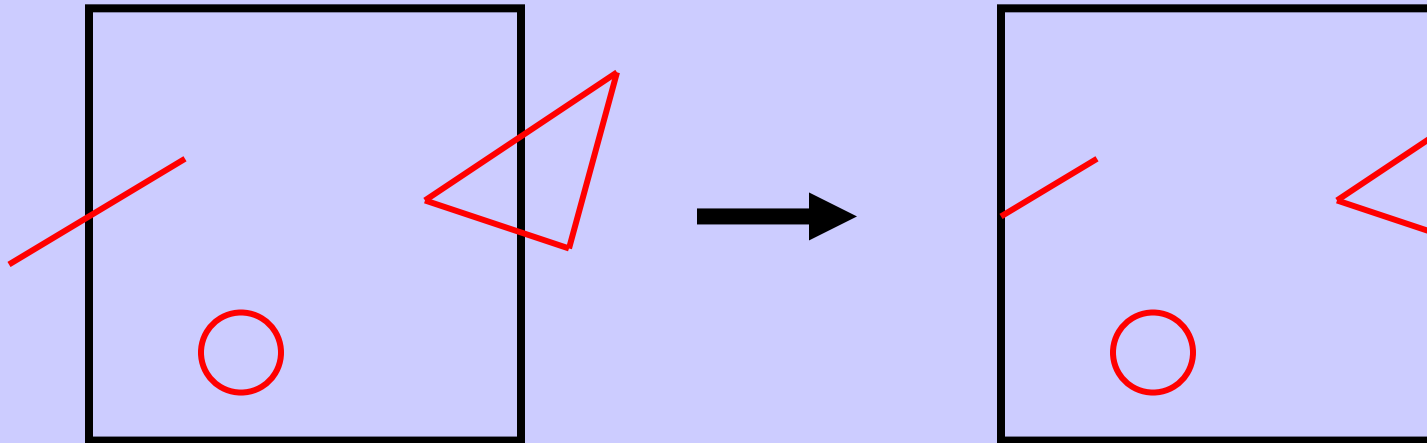
Application

Geometry

Rasterizer

## Clipping and Screen Mapping

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



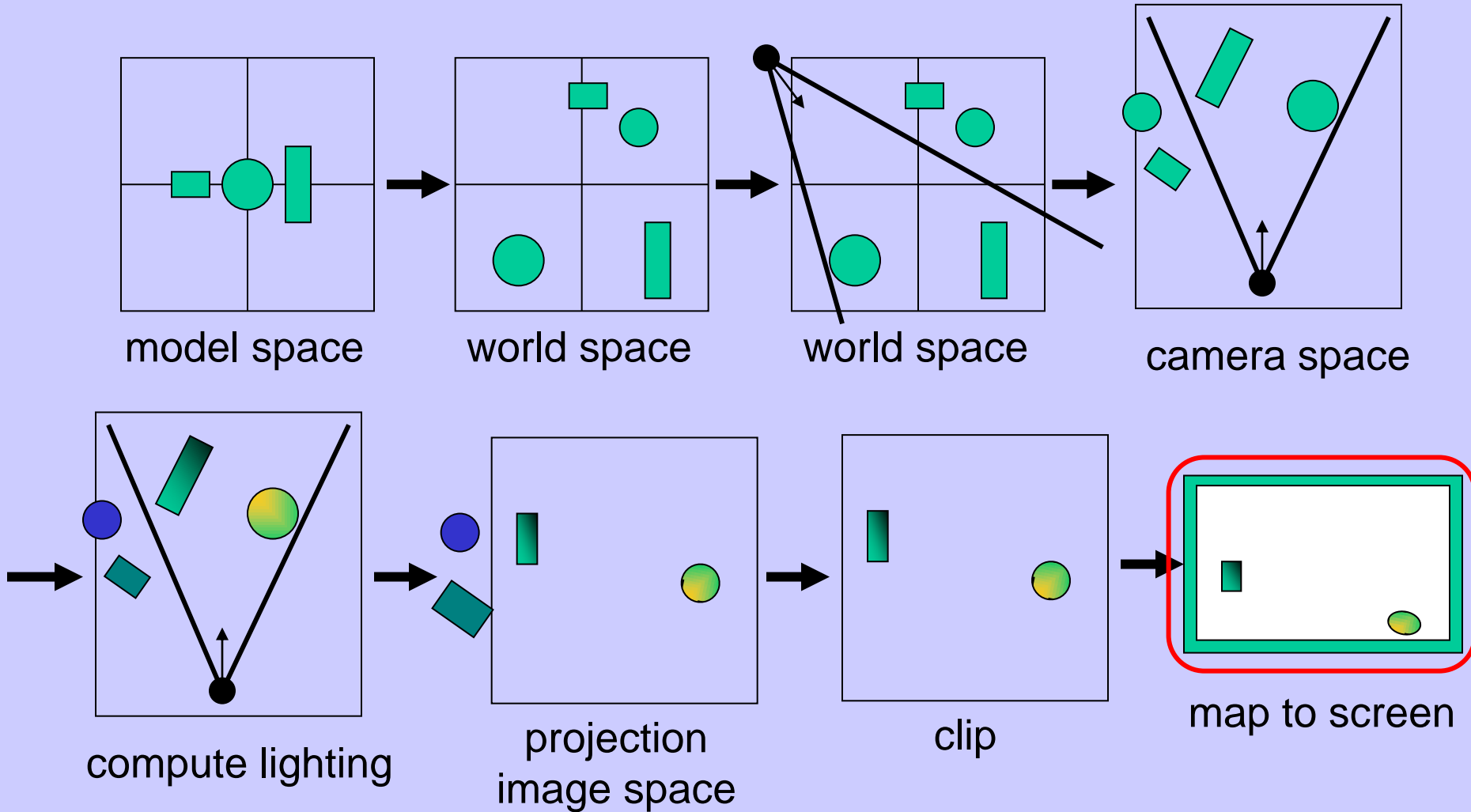
- Screen mapping, scales and translates square so that it ends up in a rendering window
- These "screen space coordinates" together with Z (depth) are sent to the rasterizer stage

Application

Geometry

Rasterizer

# GEOMETRY - Summary

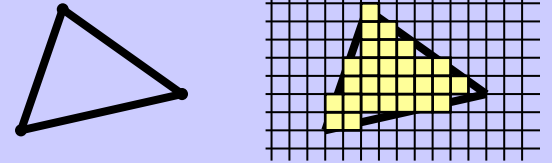


# The RASTERIZER

## in more detail

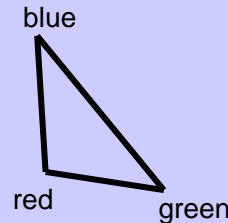
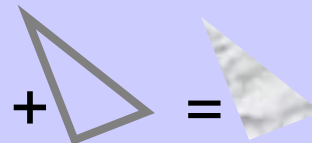
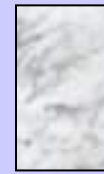
- Scan-conversion

- 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 RASTERIZER

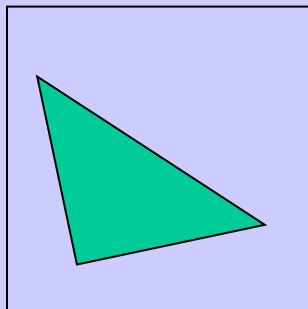
Application

Geometry

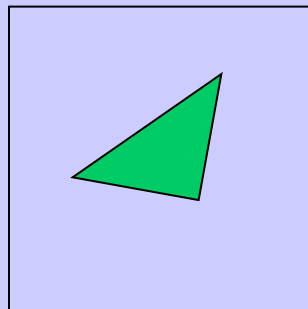
Rasterizer

## Z-buffering

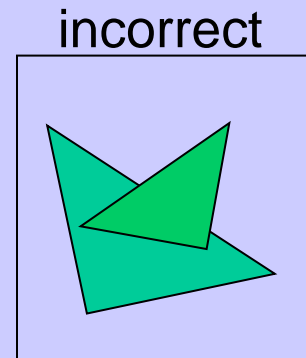
- A triangle that is covered by a more closely located triangle should not be visible
- Assume two equally large tris at different depths



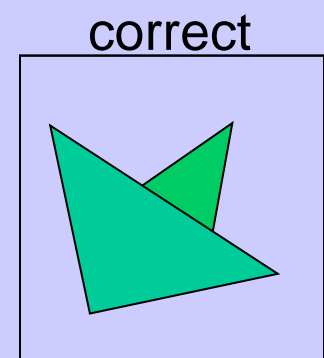
Triangle 1



Triangle 2



Draw 1 then 2



Draw 2 then 1

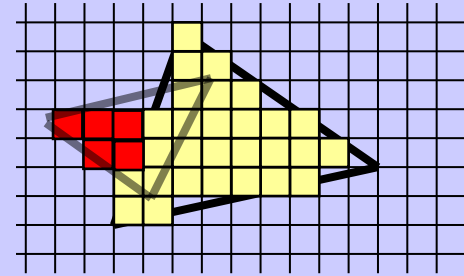
# The RASTERIZER

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

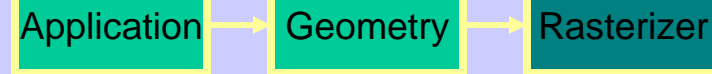
Application

Geometry

Rasterizer

## double-buffering

- The monitor displays one image at a time
- Top of screen – new image  
Bottom – old image  
No control of split position
- And even worse, we often clear the screen before generating a new image
- A better solution is "double buffering"
  - (Could instead keep track of rasterpos and vblank).



# The RASTERIZER

## double-buffering

- Use two buffers: one front and one back
- The front buffer is displayed
- The back buffer is rendered to
- When new image has been created in back buffer, swap front and back

OpenGL

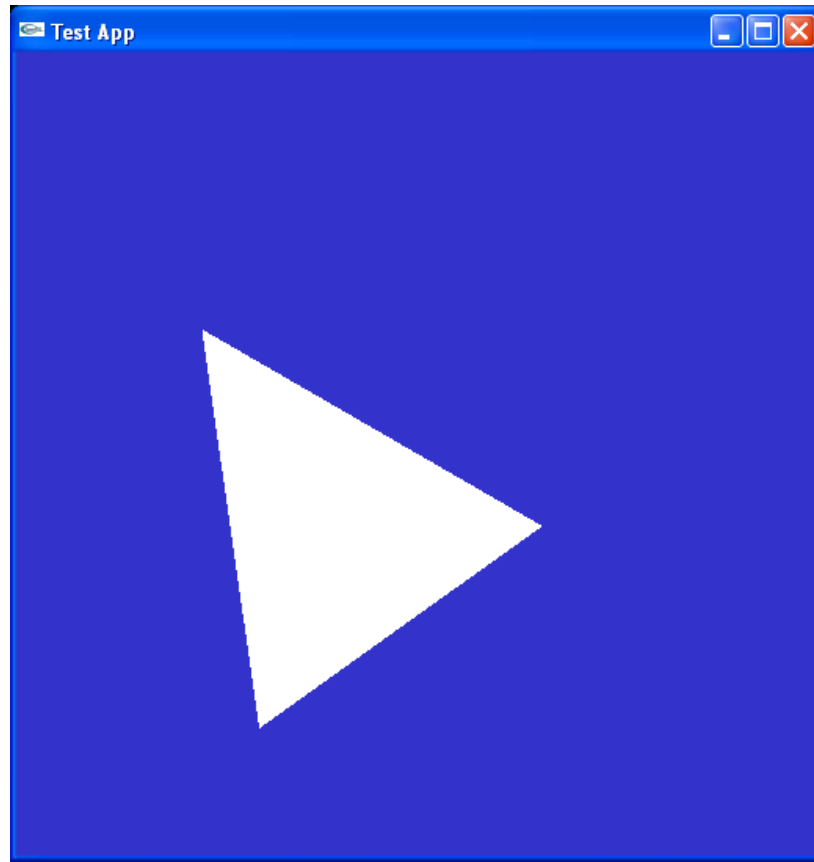


# A Simple Program

Computer Graphics version of

“Hello World”

Generate a triangle on a solid background



# Simple Application...

```
int main(int argc, char *argv[])
{
    glutInit(&argc, argv);

    /* open window of size 512x512 with double buffering, RGB colors, and Z-
    buffering */
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(512,512);
    glutCreateWindow("Test App");

    /* the display function is called once when the gluMainLoop is called,
    * but also each time the window has to be redrawn due to window
    * changes (overlap, resize, etc). */
    glutDisplayFunc(display);      // Set the main redraw function

    glutMainLoop(); /* start the program main loop */
    return 0;
}
```

```
void display(void)
{
    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

    int w = glutGet((GLenum)GLUT_WINDOW_WIDTH);
    int h = glutGet((GLenum)GLUT_WINDOW_HEIGHT);
    glViewport(0, 0, w, h);          // Set viewport

    glDisable(GL_CULL_FACE);
    drawScene();

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

```
static void drawScene(void)
{
    // Shader Program
    glUseProgramObjectARB( shaderProgram ); // Set the shader program to use for this draw call
    CHECK_GL_ERROR();

    glBindVertexArray(vertexArrayObject);
    CHECK_GL_ERROR();

    glDrawArrays( GL_TRIANGLES, 0, 3 ); // Render three points with the current sources
    CHECK_GL_ERROR();
}
```

# Shaders

```
// Vertex Shader
#version 130

in vec3 vertex;
in vec3 color;
out vec3 outColor;
uniform mat4 modelViewProjectionMatrix;

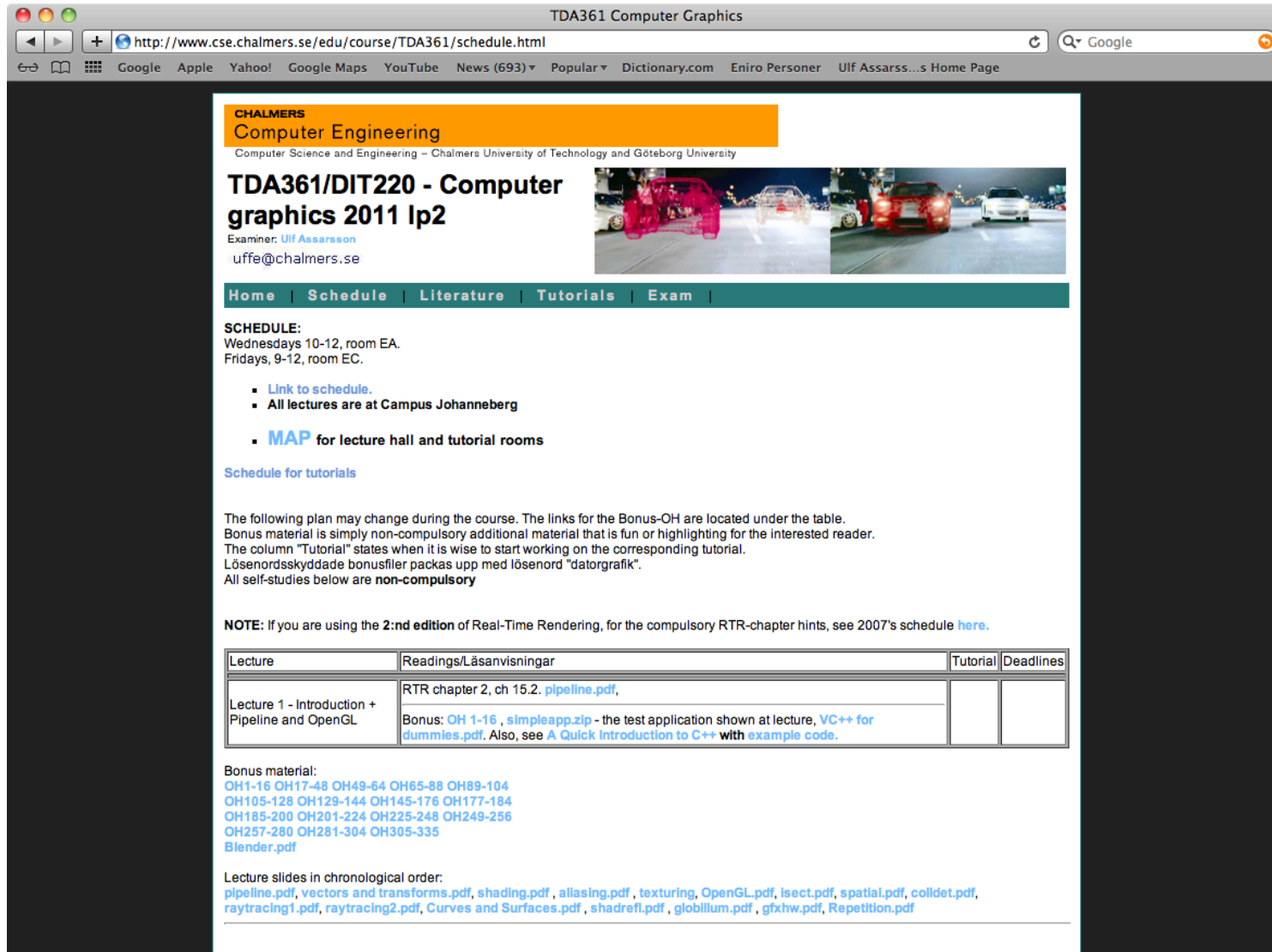
void main()
{
    gl_Position = modelViewProjectionMatrix*vec4(vertex,1);
    outColor = color;
}
```

```
// Fragment Shader:
#version 130
in vec3 outColor;
out vec4 fragColor;

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

# Demonstration of SimpleApp


- Available on course homepage in Schedule.



CHALMERS  
Computer Engineering  
Computer Science and Engineering – Chalmers University of Technology and Göteborg University

## TDA361/DIT220 - Computer graphics 2011 Ip2

Examiner: [Ulf Assarsson](#)  
[uffe@chalmers.se](mailto:uffe@chalmers.se)



[Home](#) | [Schedule](#) | [Literature](#) | [Tutorials](#) | [Exam](#)

**SCHEDULE:**  
Wednesdays 10-12, room EA.  
Fridays, 9-12, room EC.

- [Link to schedule.](#)
- All lectures are at Campus Johanneberg
- [MAP](#) for lecture hall and tutorial rooms

[Schedule for tutorials](#)

The following plan may change during the course. The links for the Bonus-OH are located under the table. Bonus material is simply non-compulsory additional material that is fun or highlighting for the interested reader. The column "Tutorial" states when it is wise to start working on the corresponding tutorial. Lösenordsskyddade bonusfiler packas upp med lösenord "datografik". All self-studies below are **non-compulsory**

**NOTE:** If you are using the 2:nd edition of Real-Time Rendering, for the compulsory RTR-chapter hints, see 2007's schedule [here](#).

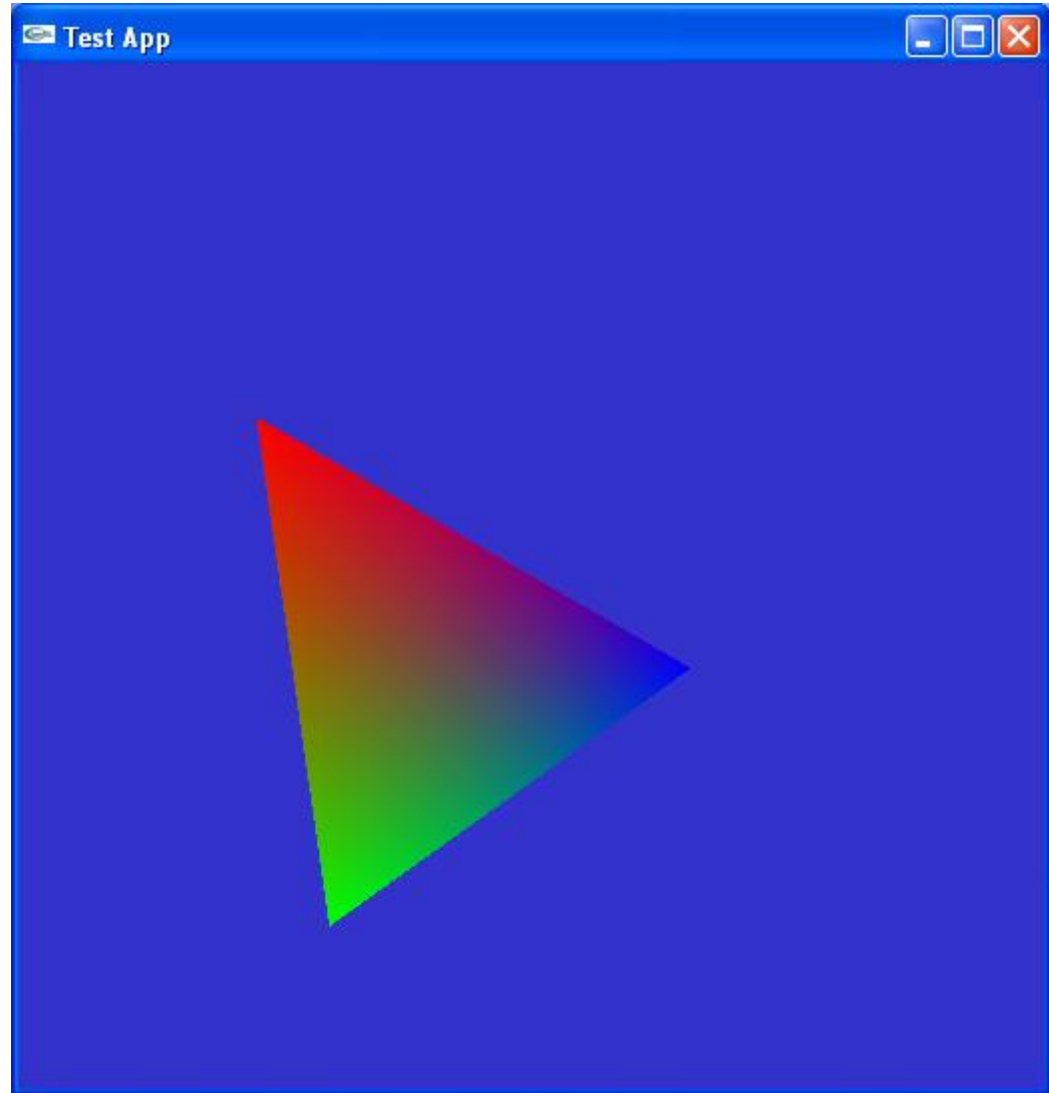
Lecture	Readings/Läsanvisningar	Tutorial	Deadlines
Lecture 1 - Introduction + Pipeline and OpenGL	RTR chapter 2, ch 15.2. <a href="#">pipeline.pdf</a> , Bonus: <a href="#">OH 1-16</a> , <a href="#">simpleapp.zip</a> - the test application shown at lecture, <a href="#">VC++ for dummies.pdf</a> . Also, see <a href="#">A Quick Introduction to C++ with example code</a> .		

Bonus material:  
[OH1-16](#) [OH17-48](#) [OH49-64](#) [OH65-88](#) [OH89-104](#)  
[OH105-128](#) [OH129-144](#) [OH145-176](#) [OH177-184](#)  
[OH185-200](#) [OH201-224](#) [OH225-248](#) [OH249-256](#)  
[OH257-280](#) [OH281-304](#) [OH305-335](#)  
[Blender.pdf](#)

Lecture slides in chronological order:  
[pipeline.pdf](#), [vectors and transforms.pdf](#), [shading.pdf](#), [aliasing.pdf](#), [texturing](#), [OpenGL.pdf](#), [isect.pdf](#), [spatial.pdf](#), [colldet.pdf](#), [raytracing1.pdf](#), [raytracing2.pdf](#), [Curves and Surfaces.pdf](#), [shadrefl.pdf](#), [globillum.pdf](#), [gfxhw.pdf](#), [Repetition.pdf](#)

# Cooler application

Starts  
looking  
good!



# Repetition

- What is important:
  - Understand the Application-, Geometry- and Rasterization Stage