CG – T4 - Representing geometric objects in 3D

L:CC, MI:ERSI

Miguel Tavares Coimbra

(course and slides designed by Verónica Costa Orvalho)

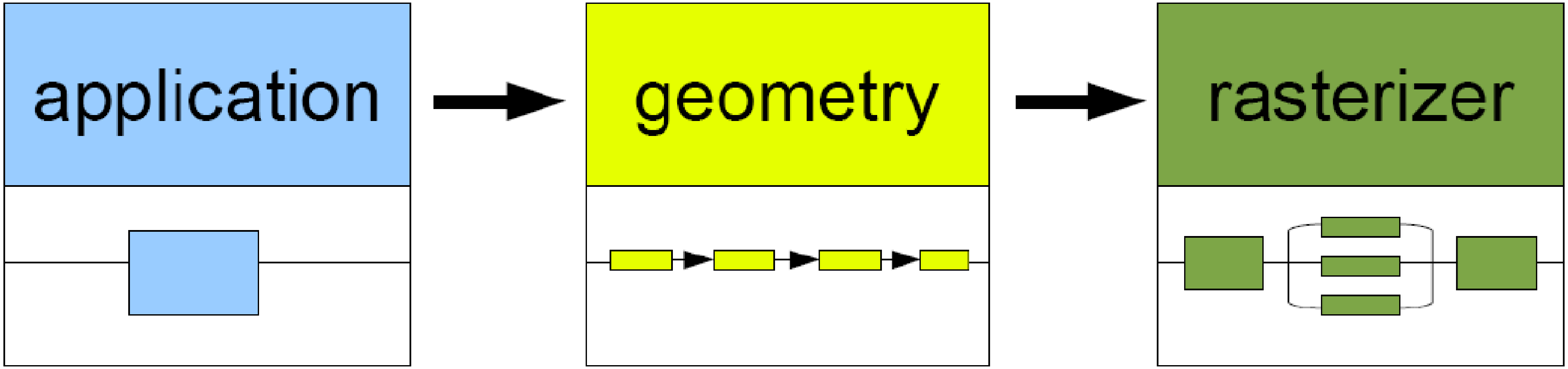
U.PORTO FC

CG Pipeline

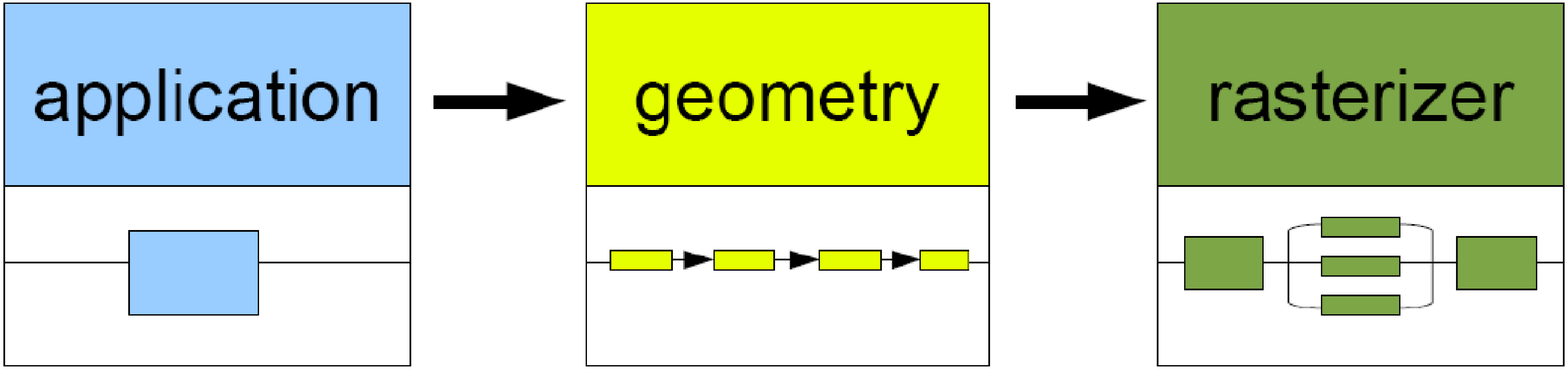
Basic steps for creating a 2D image out of a 3D world

- Create the 3D world
  - Vertexes and triangles in a 3D space
- Project it to a 2D ‘camera’
  - Use perspective to transform coordinates into a 2D space
- Paint each pixel of the 2D image
  - Rasterization, shading, texturing
  - Will break this into smaller things later on
- Enjoy the super cool image you have created

pipeline



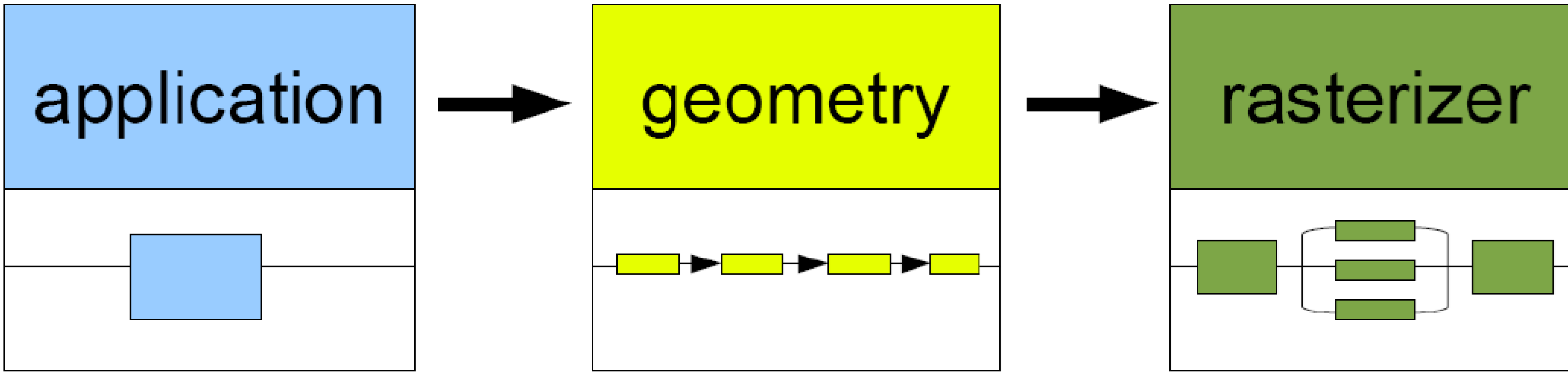
pipeline



. **collision** detection
. **animation** global acceleration
. **physics** simulation

process on CPU or GPU

pipeline



. **collision** detection
. **animation** global acceleration
. **physics** simulation

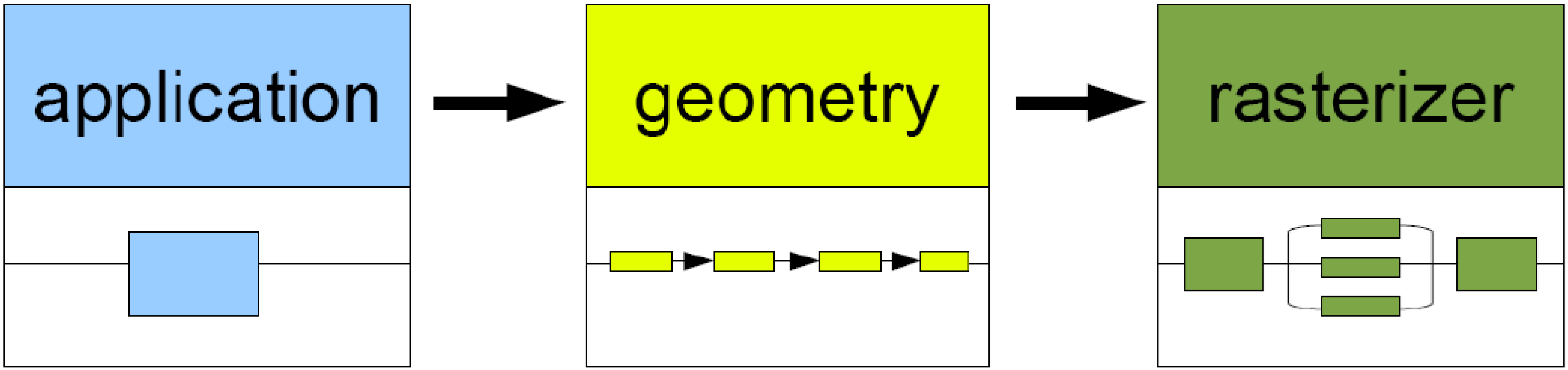
process on CPU or GPU

. **transformation**
. **projection**

Computes:
. what is to be drawn
. how should be drawn
. where should be drawn

process on GPU

pipeline



. **collision** detection
. **animation** global acceleration
. **physics** simulation

process on CPU or GPU

. **transformation**
. **projection**

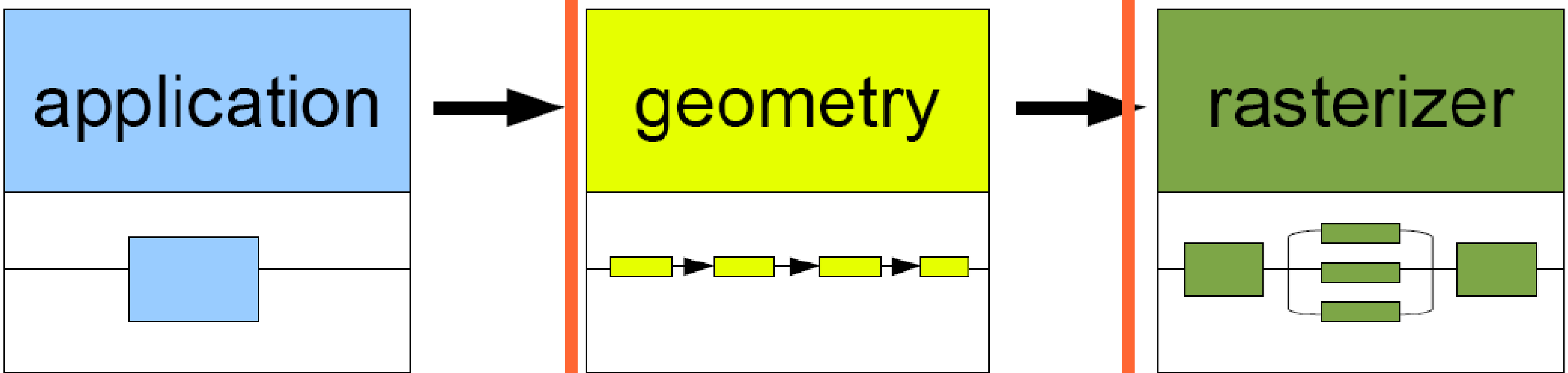
Computes:
. what is to be drawn
. how should be drawn
. where should be drawn

process on GPU

. **draws** images generated by **geometry stage**

process on GPU

pipeline



. **collision** detection
. **animation** global acceleration
. **physics** simulation

process on CPU or GPU

. **transformation**
. **projection**

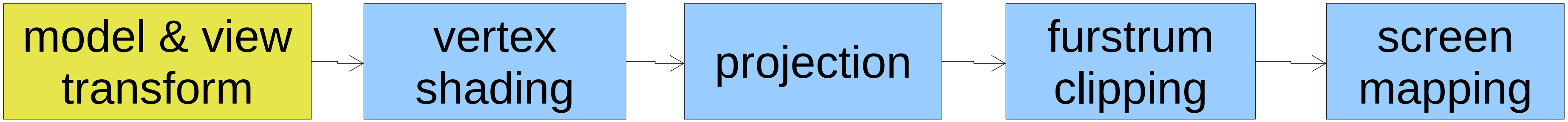
Computes:
. what is to be drawn
. how should be drawn
. where should be drawn

process on GPU

. **draws** images generated by **geometry stage**

process on GPU

pipeline **REVIEW**

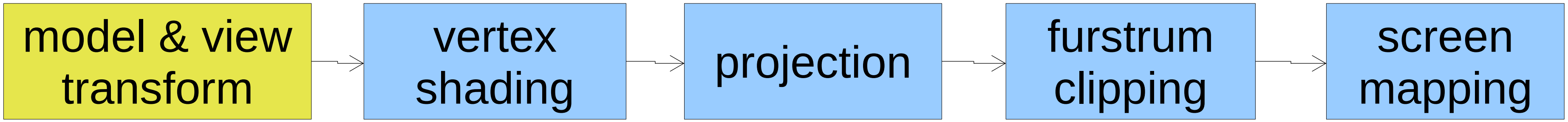


transformation
from:
<u>model coord.</u>
to
<u>world coord.</u>

↓

delinated by
__?__ space

pipeline **REVIEW**



transformation
from:
model coord.
to
world coord.

↓

delinated by camera space

pipeline **REVIEW**

model & view transform → vertex shading → projection → furstrum clipping → screen mapping

transformation from:
<u>model coord.</u>
to
<u>world coord.</u>

↓

delinated by
<u>camera space</u>

effect of a light on a material

↓

computed in
__?__ space

pipeline **REVIEW**

model & view transform → vertex shading → projection → furstrum clipping → screen mapping

transformation from:
model coord.
to
world coord.

↓

delinated by camera space

effect of a light on a material

↓

computed in world space

U.PORTO FC

pipeline **REVIEW**

model & view transform	vertex shading	projection	furstrum clipping	screen mapping

transformation from:
model coord.
to
world coord.

↓

delinated by camera space

effect of a light on a material

↓

computed in world space

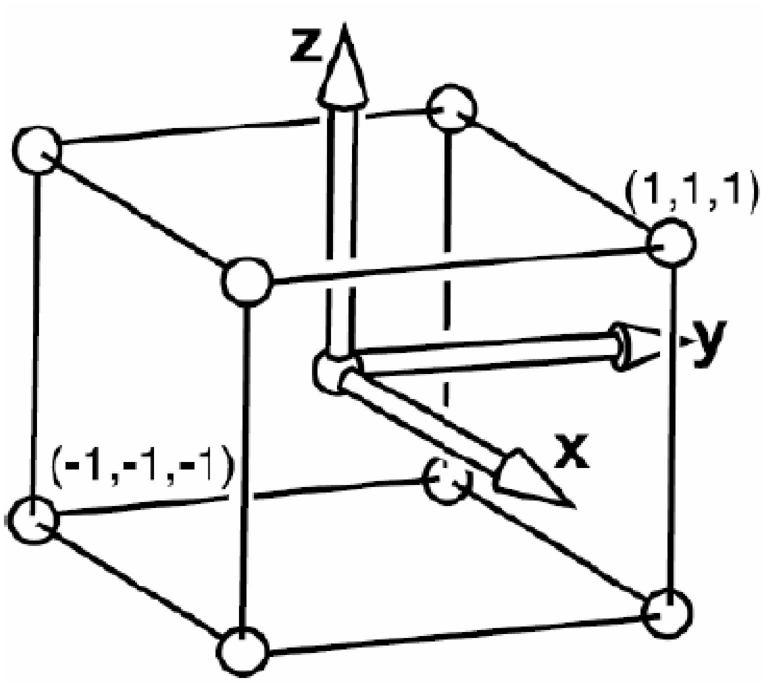
transform the
view volume
into a
unite cube

↓

call canonical view volume

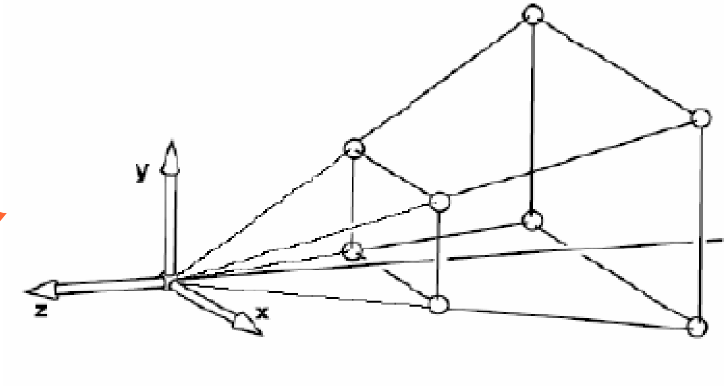
pipeline **REVIEW**

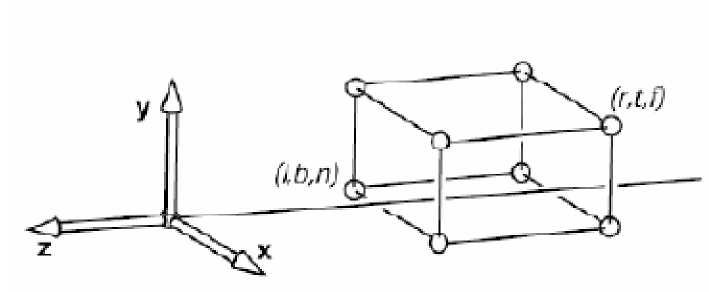
model & view transform → vertex shading → projection → furstrum clipping → screen mapping



canonical view volume

perspective projection

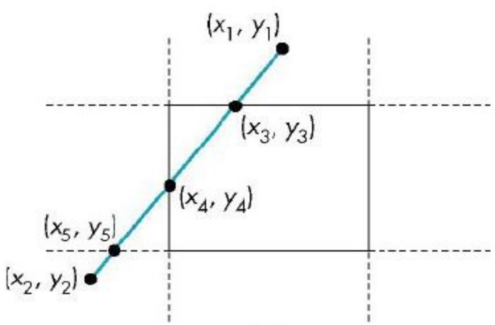




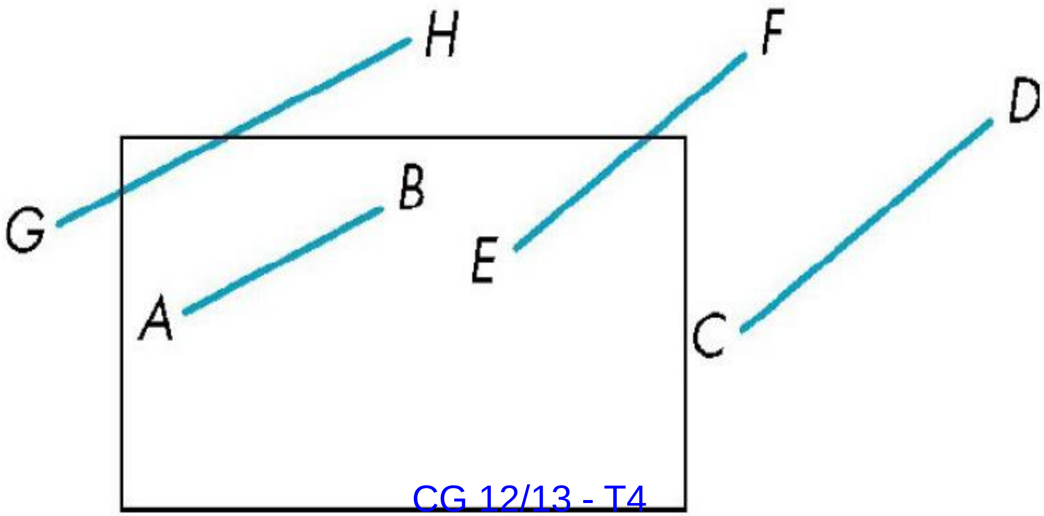
orthographic projection

pipeline **REVIEW**

model & view transform → vertex shading → projection → furstrum clipping → screen mapping



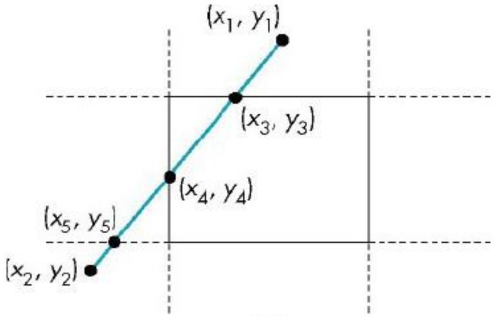
draw primitives that are inside the **__?__volume**

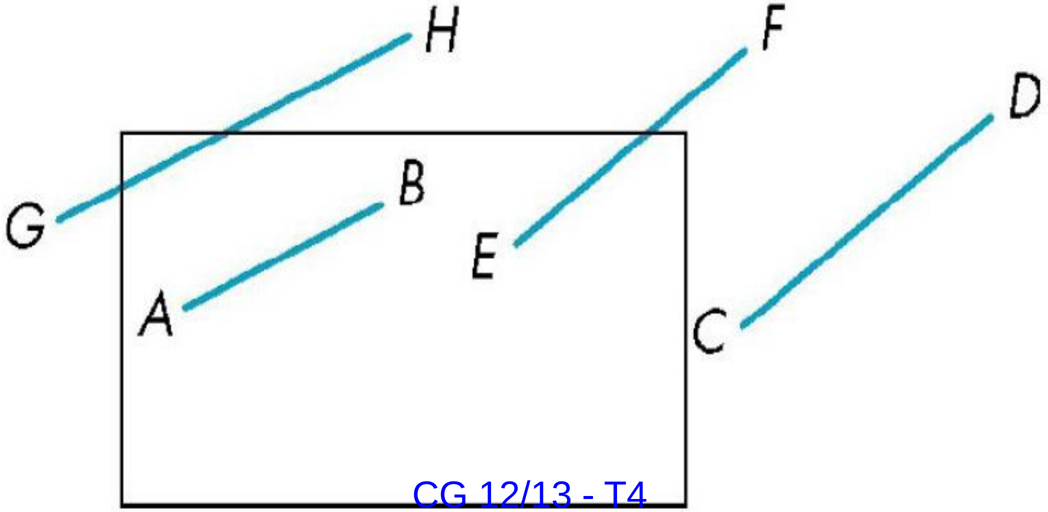


perform in HW

pipeline **REVIEW**

model & view transform → vertex shading → projection → furstrum clipping → screen mapping





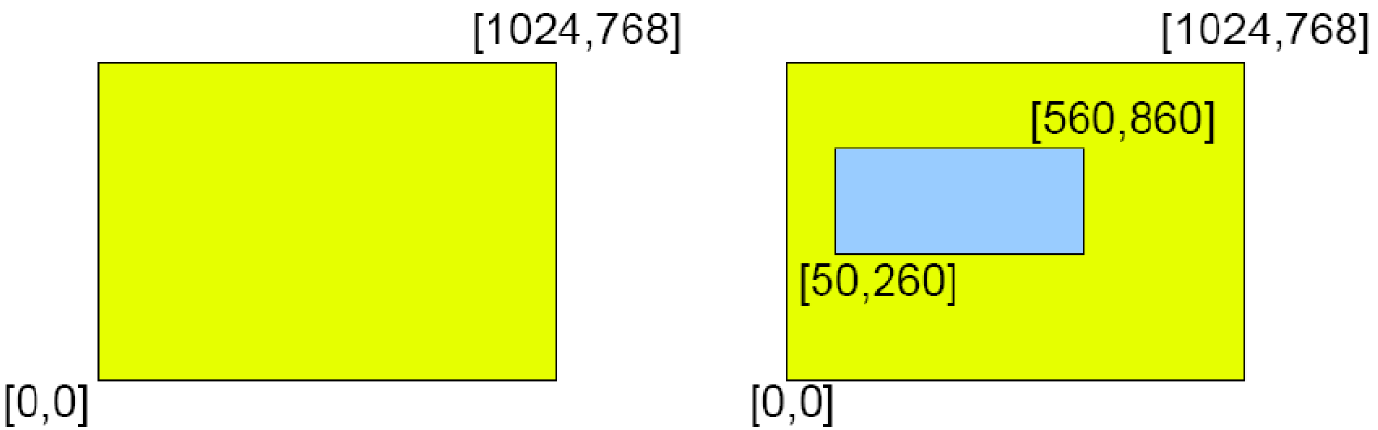
draw primitives that are inside the **view volume**

↓

perform in HW

pipeline **REVIEW**

transform to **window coord**

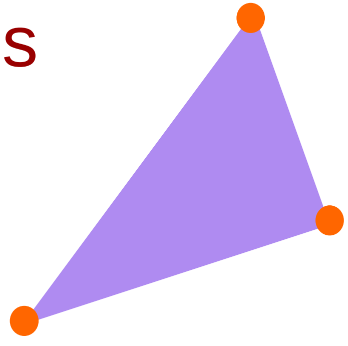


you can have more than 1 viewport

3D Objects

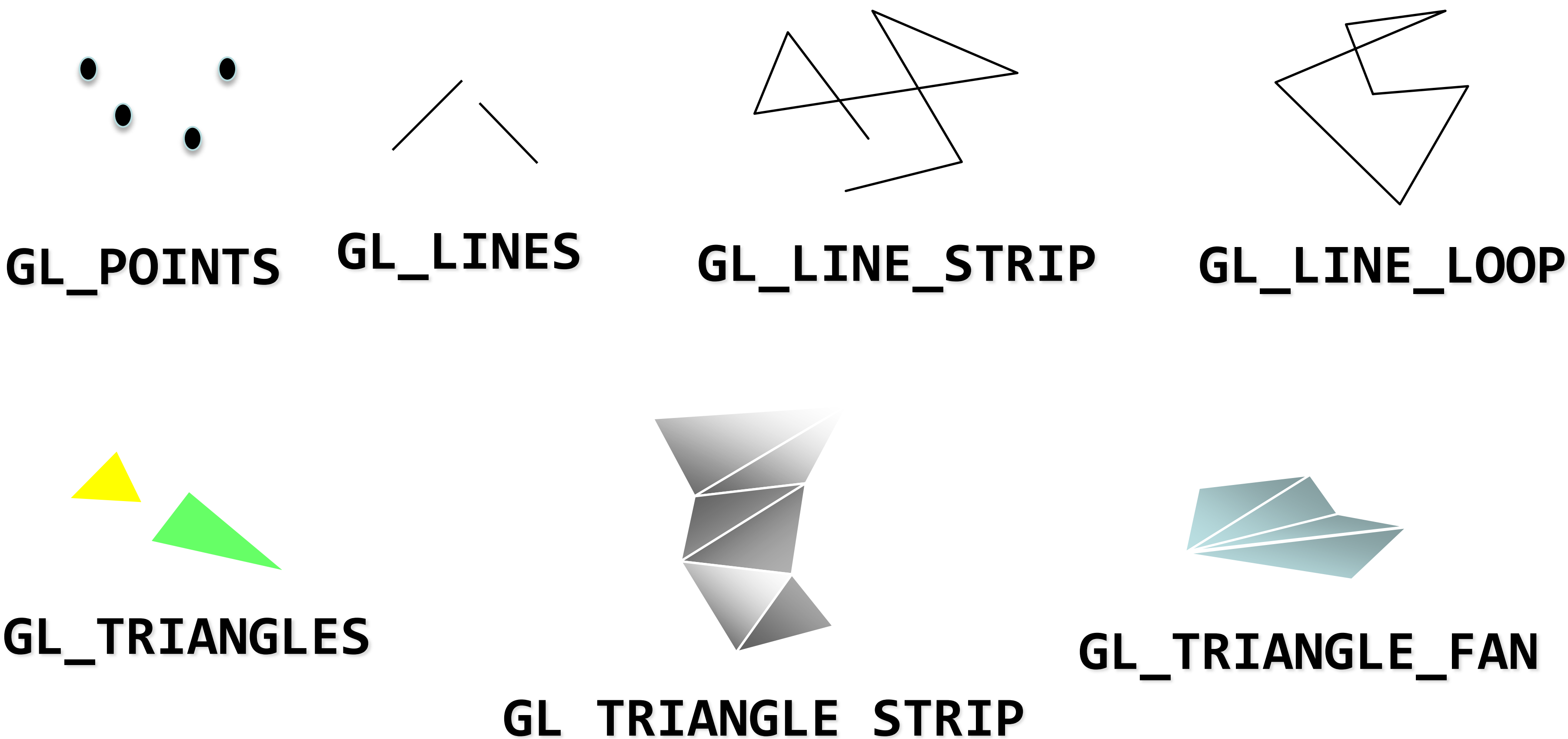
Representing Geometric Objects

- Geometric objects are represented using vertices
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space
- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in *vertex buffer objects* (VBOs)
- VBOs must be stored in *vertex array objects* (VAOs)



OpenGL's Geometric Primitives

All primitives are specified by vertices



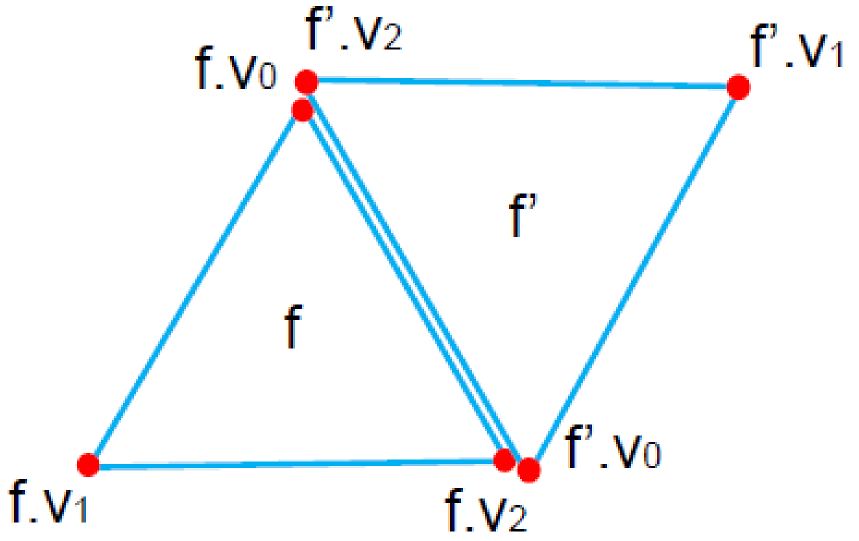
What should be a good data structure for storing my triangles?

- Various options
  - Can you describe some of them?
- An efficient one:
  - Store vertexs in its own data structure
    - In OpenGL: VBO – vertex buffer objects
  - Store triangles (objects) in its own data structure
    - In OpenGL: VAO - *vertex array objects*

Data Structure: Separate Triangles

Treat each triangle separately with its own vertices

```
typedef float Point[3];
struct Face {
    Point v[3];
};
mesh = Face[nFaces];
```



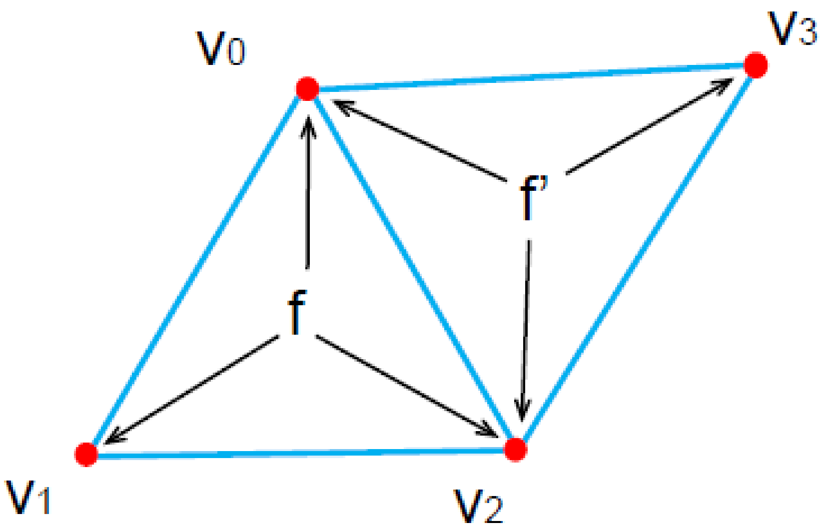
Storage: 72 bytes per vertex

No notion of "neighbor triangles": Individual triangles might not overlap with their vertices or edges

Data Structure: Indexed Triangle Set

Store each vertex only once; each face contains indices to its three vertices

```
typedef float Point[3];
struct Face {
     int vIndex[3];
};
mesh.verts=Point[nVerts];
mesh.faces=Face[nFaces];
```



Storage: 12 (verts) + 24 (faces) = 36 bytes per vertex
(approximate using #f = 2 #v)

By removing vertex redundancy we have a notion of neighbor, however finding any neighbor requires a global search

Slide by Ron Fedkiw, Stanford University

Comparison

Separate Triangles (Vertex Buffer only)

- + Simple
- – Redundant information

Indexed Triangle Set (Vertex Buffer + Index Buffer)

- + Sharing vertices reduces memory usage
- + Ensure integrity of the mesh (moving a vertex causes that vertex in all the polygons to be moved)

+ Both formats are compact and directly accepted by GPUs

+ Both can represent non-manifold meshes

– Neither is good at neighborhood access/modification

Slide by Ron Fedkiw, Stanford University

Example: Storing a Cube

Our First Program

- We'll render a cube with colors at each vertex
- Our example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices

Initializing the Cube's Data

- We'll build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
const int NumVertices = 36;
```

- To simplify communicating with GLSL, we'll use a vec4 class (implemented in C++) similar to GLSL's vec4 type
  - we'll also typedef it to add logical meaning

```
typedef  vec4  point4;
typedef  vec4  color4;
```

Initializing the Cube's Data (cont'd)

- Before we can initialize our VBO, we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
point4  points[NumVertices];
color4  colors[NumVertices];
```

Cube Data

```
// Vertices of a unit cube centered at origin, sides aligned
   with axes
point4 vertex_positions[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```

Cube Data

```
// RGBA colors
color4 vertex_colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Generating a Cube Face from Vertices

```
// quad() generates two triangles for each face and assigns colors to the
   vertices
int Index = 0;  // global variable indexing into VBO arrays

void quad( int a, int b, int c, int d )
{
    colors[Index] = vertex_colors[a]; points[Index] = vertex_positions[a];
   Index++;
    colors[Index] = vertex_colors[b]; points[Index] = vertex_positions[b];
   Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_positions[c];
   Index++;
    colors[Index] = vertex_colors[a]; points[Index] = vertex_positions[a];
   Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_positions[c];
   Index++;
    colors[Index] = vertex_colors[d]; points[Index] = vertex_positions[d];
   Index++;
}
```

Generating the Cube from Faces

```
// generate 12 triangles: 36 vertices and 36
  colors
void
colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

What about VBOs and VAOs?

- That's what we will explore in the lab
- In the meantime:
  - Introduction to Modern OpenGL Programming
  - http://www.daveshreiner.com/SIGGRAPH/s11/