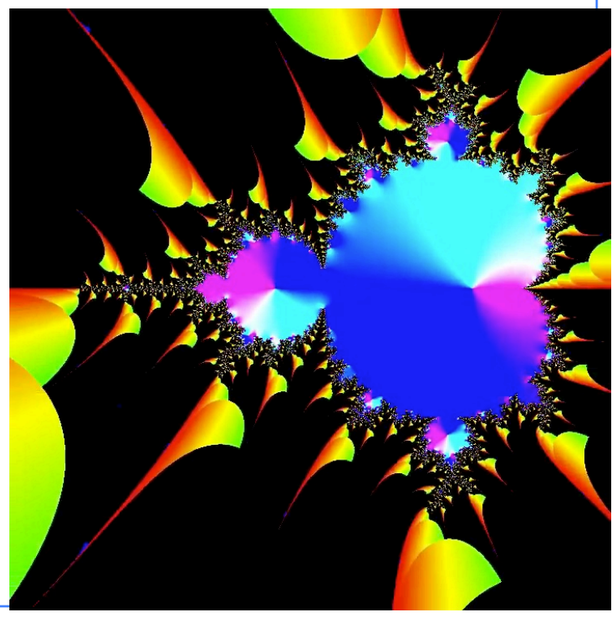
TSBK07/TSBK11/ETE378

Computer Graphics

Ingemar Ragnemalm, ISY



Lecture 9

A bit more on portals

Occlusion culling: PVS

Level of detail

Billboards

Last time

Picking

Rotation around arbitrary axis

Trackball controls

Large worlds:

Frustum culling

Occlusion culling: Portals

Lecture questions

1. What is the advantage with PVS over Portals?

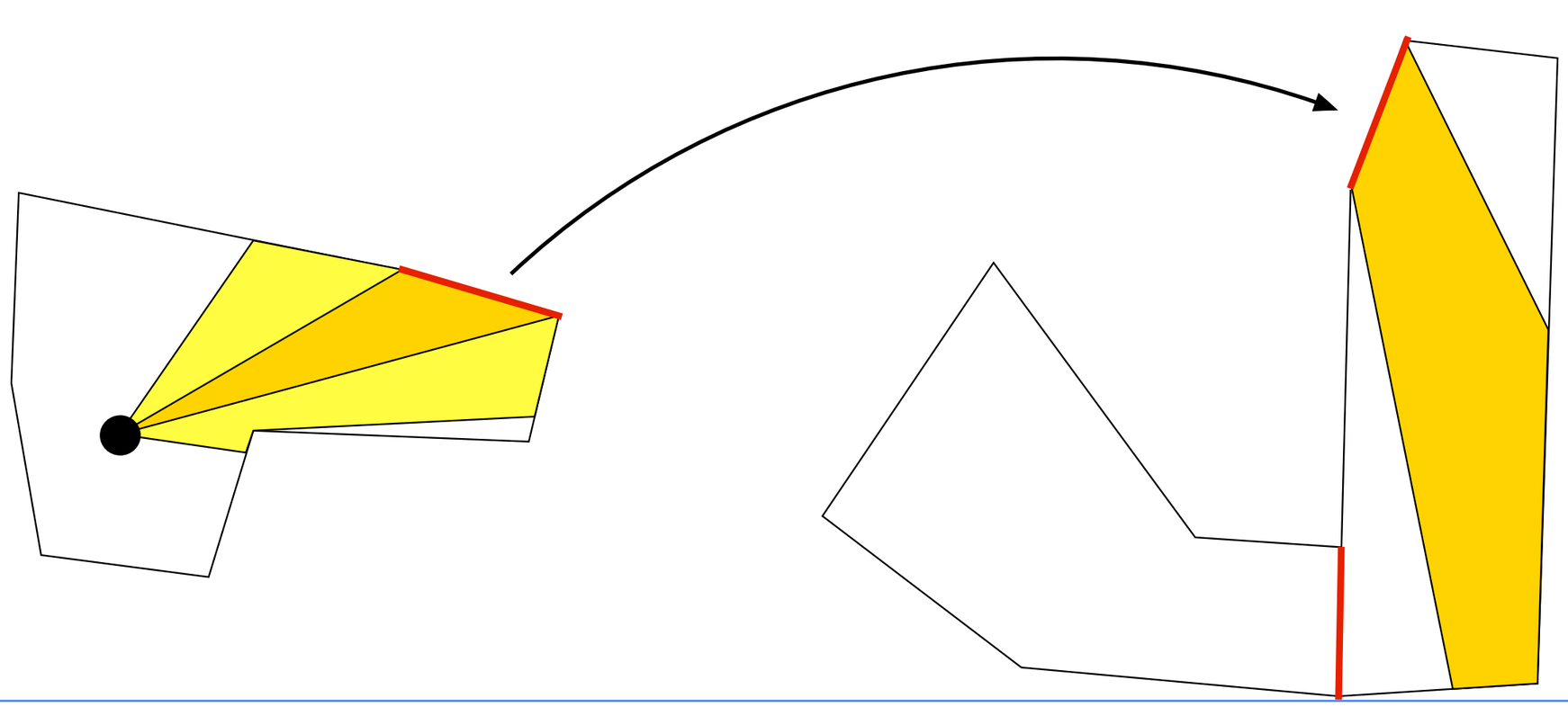
2. How can you reduce the detail of a model?

3. Which kind of billboard is easiest to implement?

4. When can you avoid sorting a particle system?

Portal transformations

Nothing stops you from putting a transformation in your portals!



Portal scenes

Conflict free environments

No overlapping scenes through multiple portals

Suitable to make with the course base.

Examples:

1) The wall. A single portal, no portal visible through the portal

2) The corridor. A repeated room. No overlaps possible.

Hard portal scenes

Overlapping scenes through different portals.

Need additional tools: Stencil buffer, scissoriing, render to texture...

Beyond the course base!

However, an alternative:

Ray-traced portals.

Portals are

- An optimization method
- A special effect method

Both are based on the same principles.

Types of portals for effects:

- Conflict free environments. Works with course material.
- Ray-tracing portals, also feasible

These need advanced course material:

- Stencil portals
- Texture portals
- FBO portals

Occlusion culling 2: Potentially visible set (PVS)

A bit list for some part of the world (cell), specifying what polygons may be visible. (Quake)

Originally a polygon list.

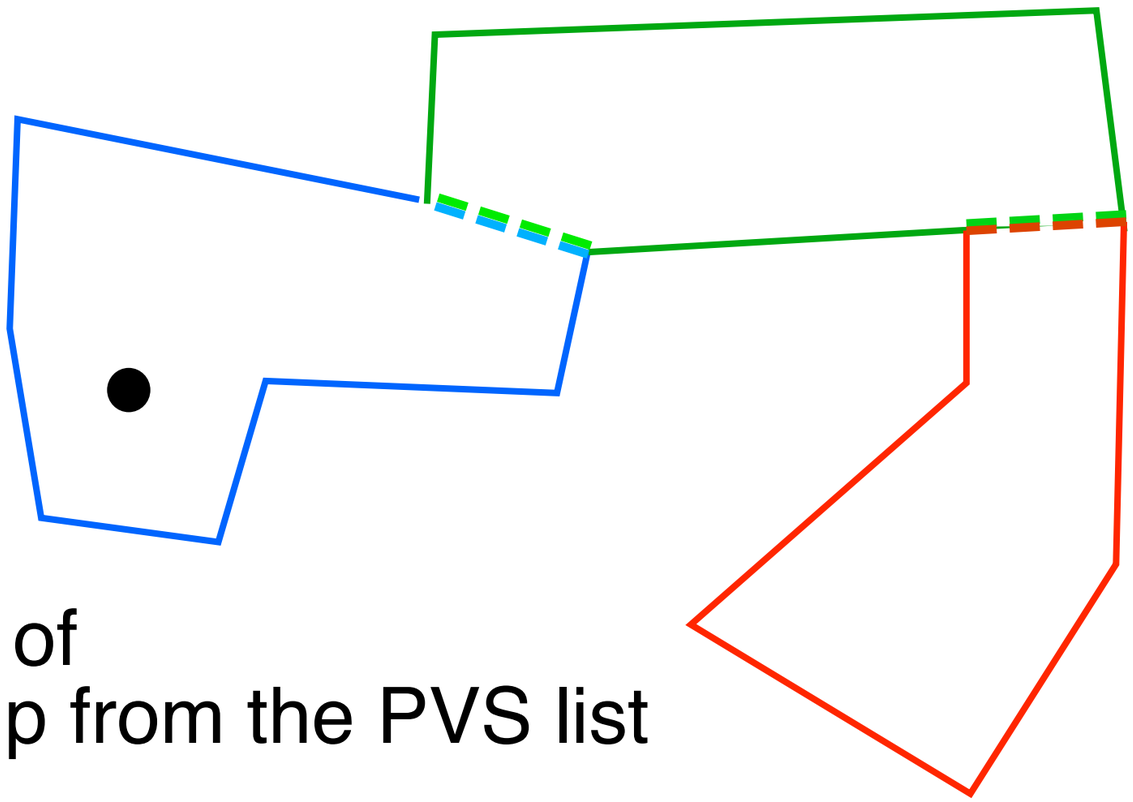
Today: Consider entire cells

Pre-compute the list for a static set of cells.

Use BSP trees for creating cells automatically.

Potentially Visible Set

More general method than portals, faster for very complex scenes.



`Blue: Can see green`

`Green: Can see blue and red`

`Red: Can see green`

Can you see when we get an error?

-> Approximative VSD method!

All cells that may be of interest are looked up from the PVS list

Pre-generating the PVS

Done for a number of points in a cell

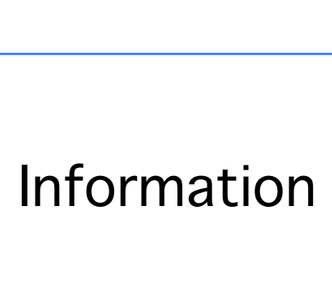
1) Image-space method

2) Object-space method

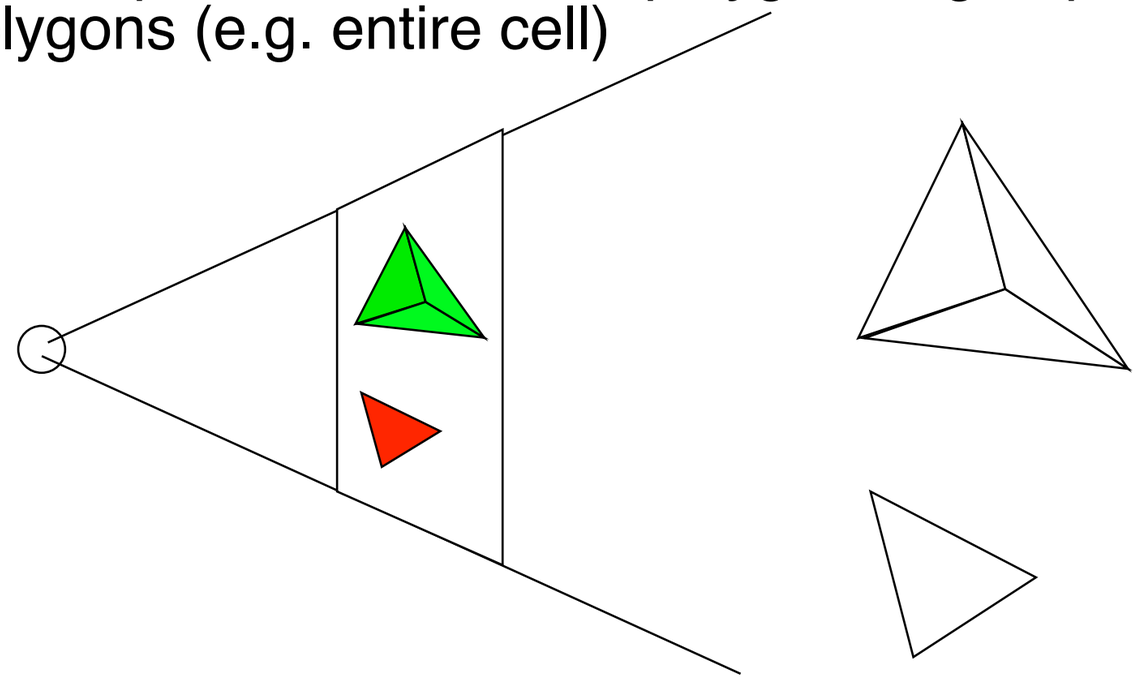
Image-space PVS generation

Render 6 images, all covering 1/6 of direction space

Render with flat shading, unique colors for each polygon or groups of polygons (e.g. entire cell)



Render to all sides of a cube around the cell



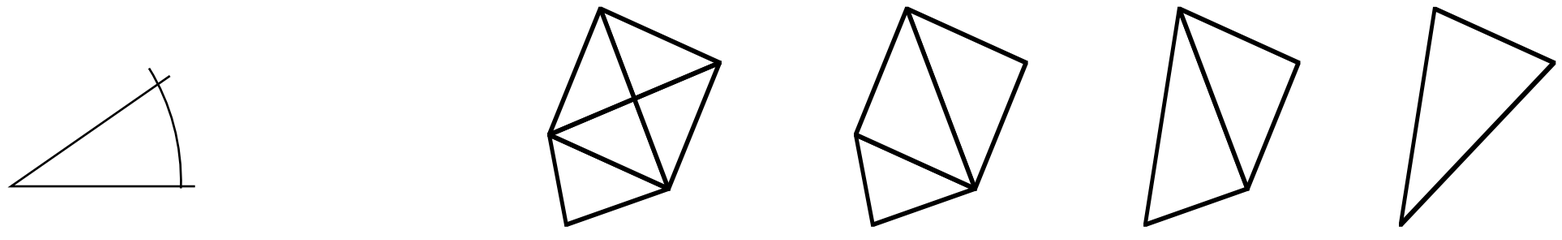
Inspect the resulting images. For every color that appears, the corresponding cell is added to the PVS

Conclusions about Visibility processing/ High level VSD:

- Frustum culling (relatively) easy!
- Doesn’t have to be perfect - some waste at edges are OK
- Occlusion culling can optimize even further

Level-of-detail LOD

Multiresolution representations
Reducing the polygon count for distant objects



Example: Stanford bunny

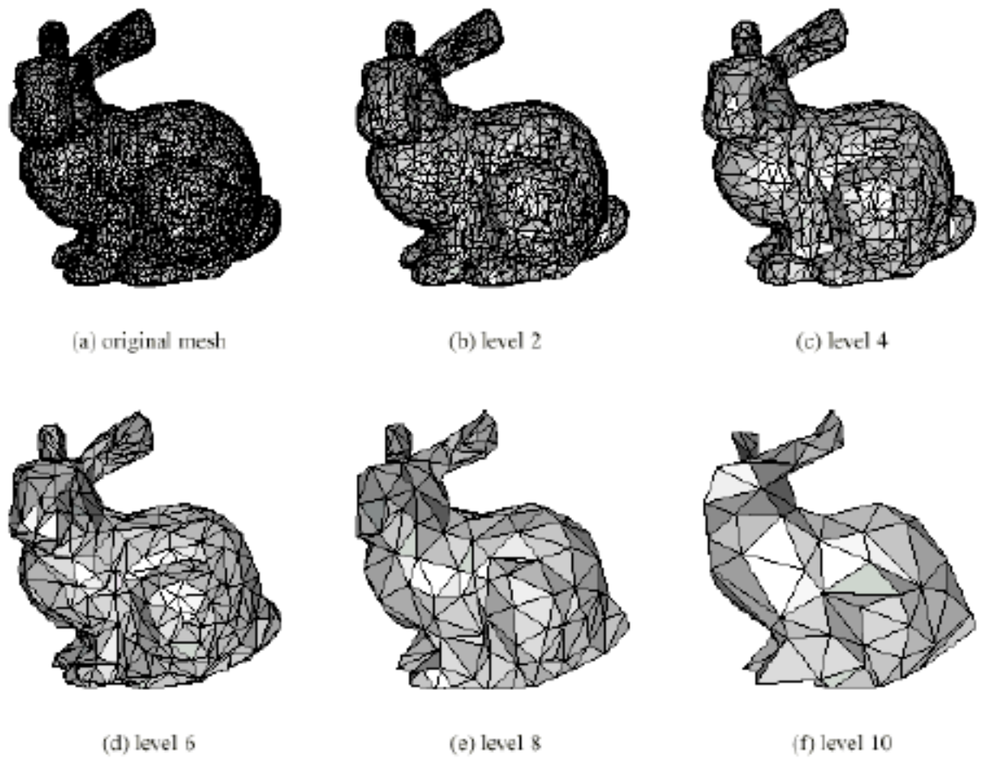


Figure 8. Stanford Bunny. Simplified meshes with 10000, 4577, 2106, 988, 463, 245 triangles

Level-of-detail LOD for models

1. Pre-generate in different detail

Risk for noticable “popping” when switching model

2. Progressive mesh

Continuous deformation, no “popping”

Non-trivial to select the polygons to reduce

At very low resolutions, we may switch to impostors (billboards)

Mesh decimation, reduction methods

Collapse edges

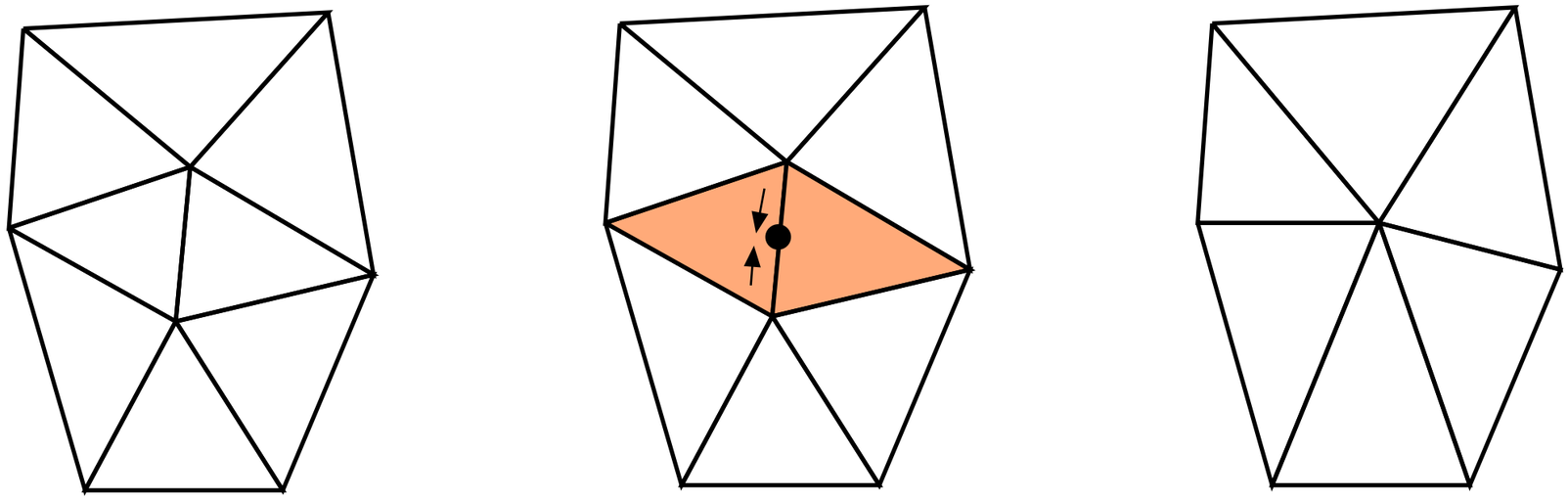
Insert new vertex, remove neighbors, re-triangulate

Remove vertices

Remove vertices, re-triangulate
(similar)

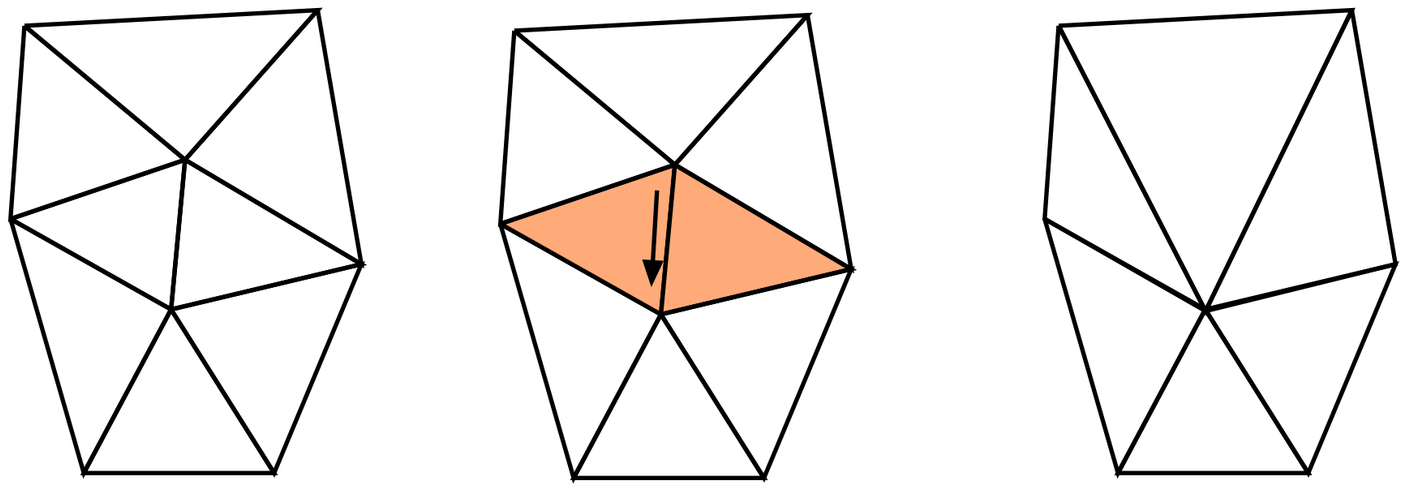
Find neighbor polygons in the same plane (or near), and merge them.

Edge collapsing



Simple - but vertex attributes (normals, texture coords) must be recalculated

Vertex removal

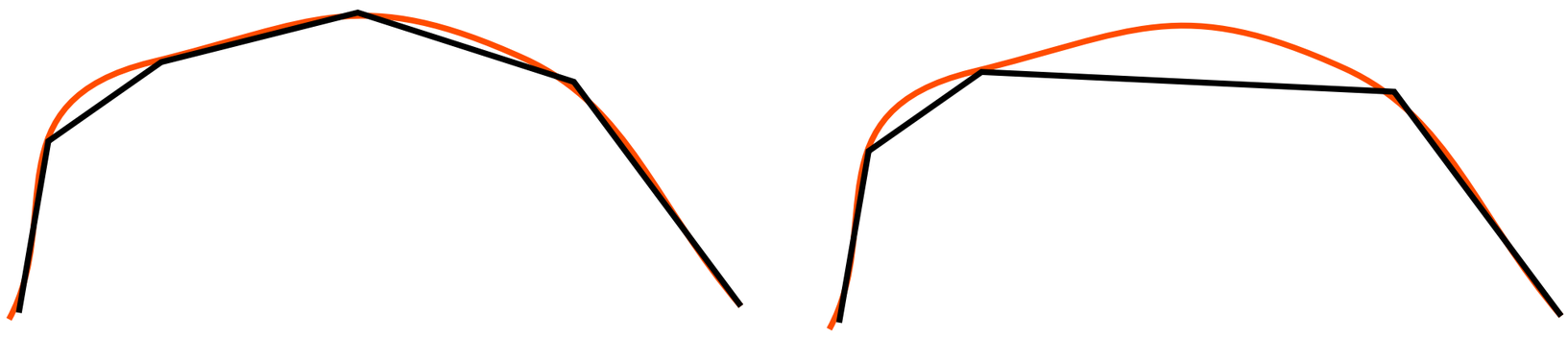


Simple - no recalculation of vertex attributes

Problem in mesh decimation: volume reduction

The mesh is a sampling of a continuous surface

Careless removal or interpolation will cause errors



Measured according to Garland & Heckbert, "error quadrics"

Further demands in mesh decimation

- Preserve high curvature
- Compensate for volume loss
- Minimize error in texture placement
- Avoid very thin polygons
- Vary decimation depending on importance of features

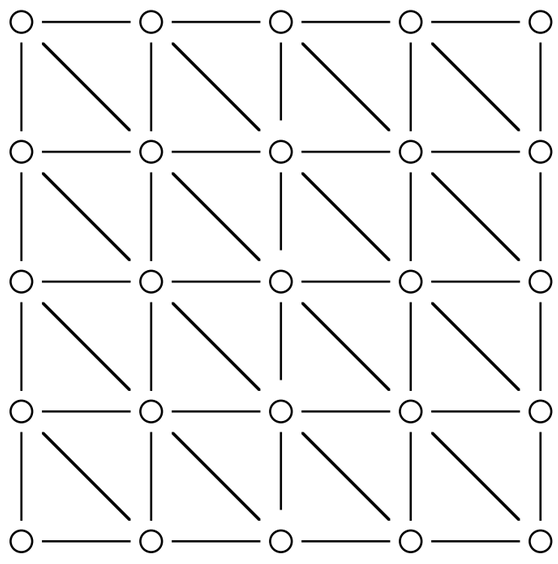
Level-of-detail LOD for terrains

Geometrical mip-mapping

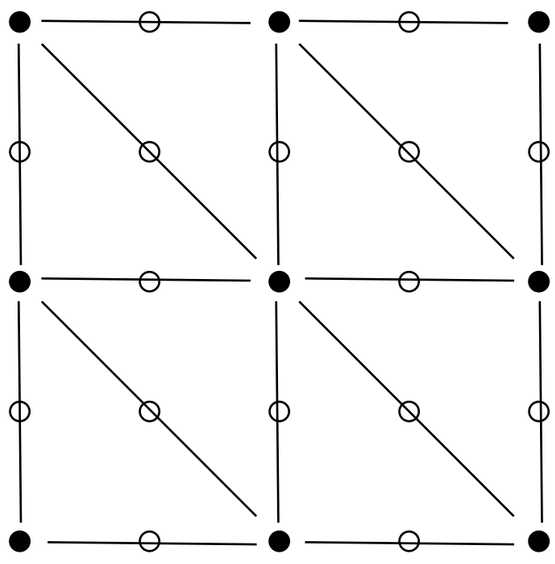
Produces a polygon terrain with approximately constant polygon size in screen coordinates

Reduces the polygon count effectively to what is actually needed.

Geometrical mip-mapping



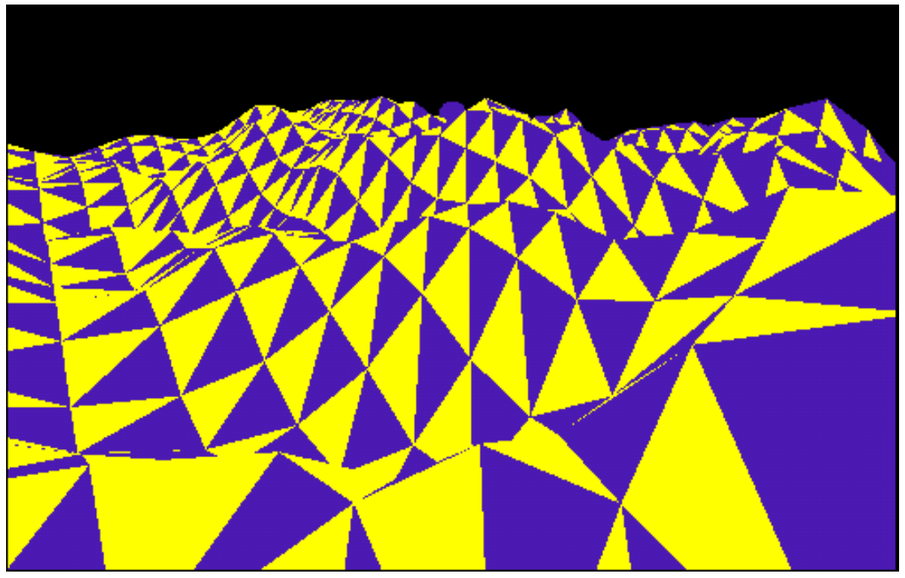
Level 0 - full resolution



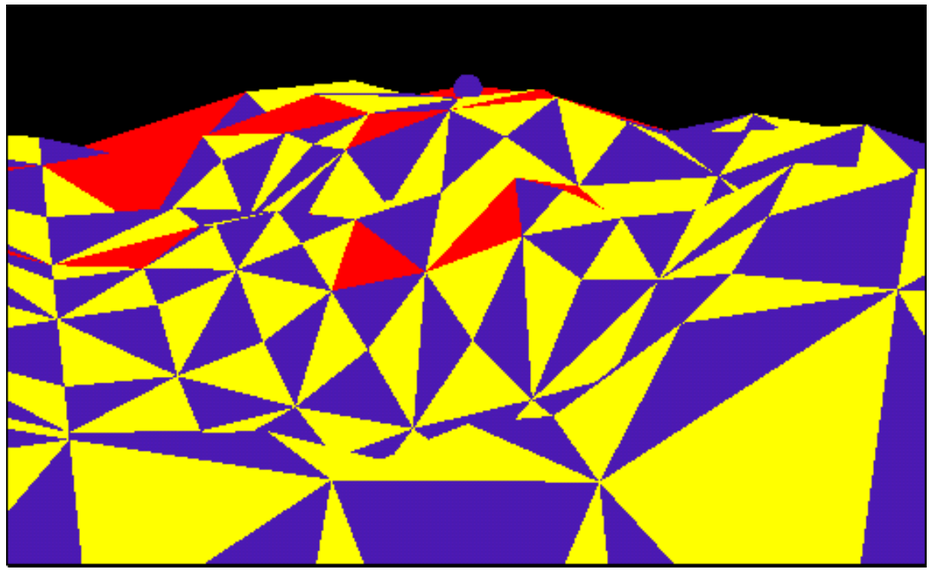
Level 1

Geometrical mip-mapping

No geomipmapping - polygon density grows with distance

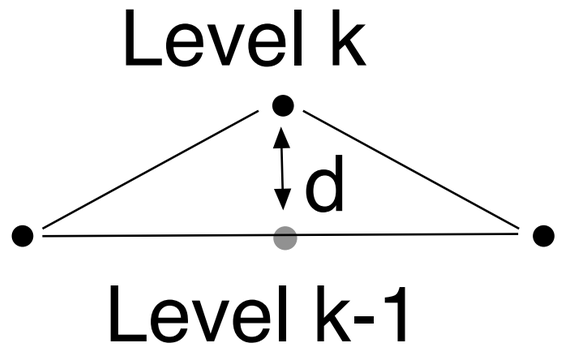


With geomipmapping - polygon density similar on all distances



Decide resolution level

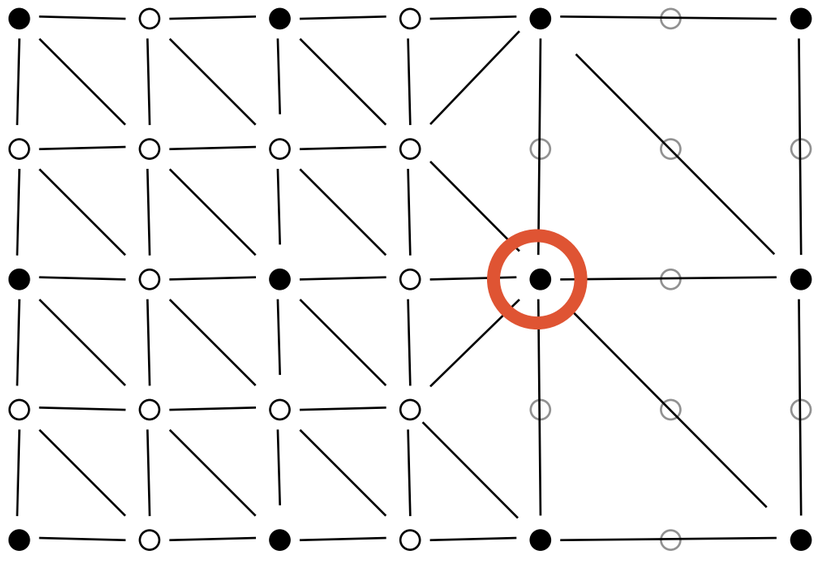
- distance
- screen-space error measures



Problems to solve in geomipmapping

- Popping
- Gaps

Patching edges between different levels

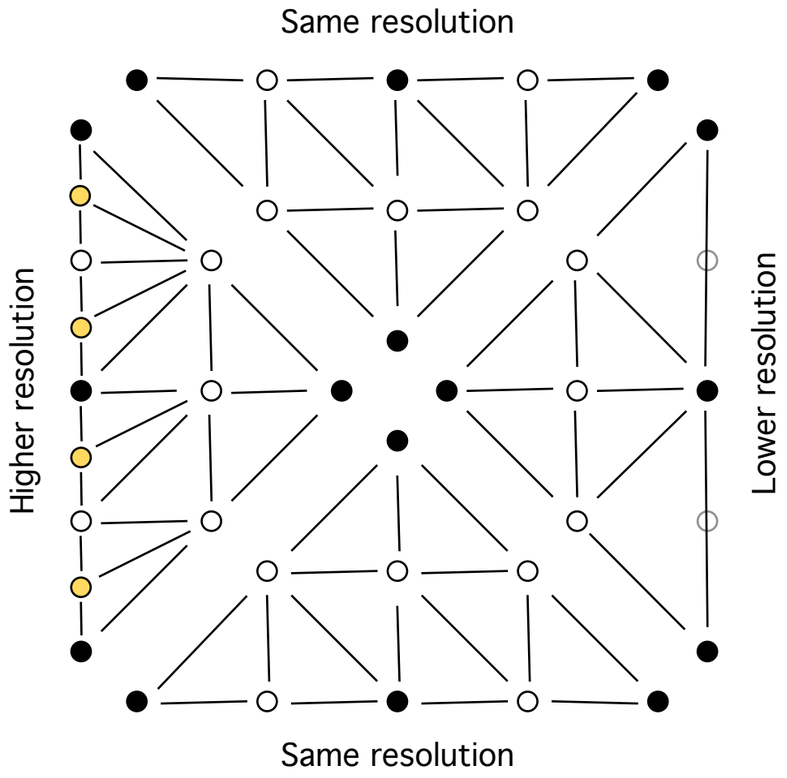


Decisions best taken at edges between patches!

Why take decisions at edges?

Both patches will take the same decision at that particular edge. What happens along other edges does not matter.

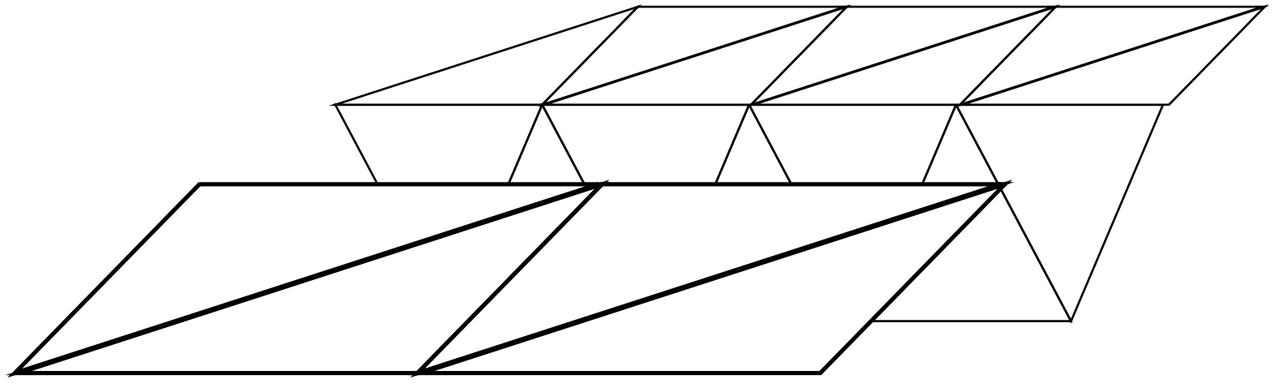
One approach: Split each patch in 4 parts with different resolution depending on the edge!



Patching edges between different levels

Second approach: "Skirts". Insert extra polygons at the edge in a way that will fill the gap.

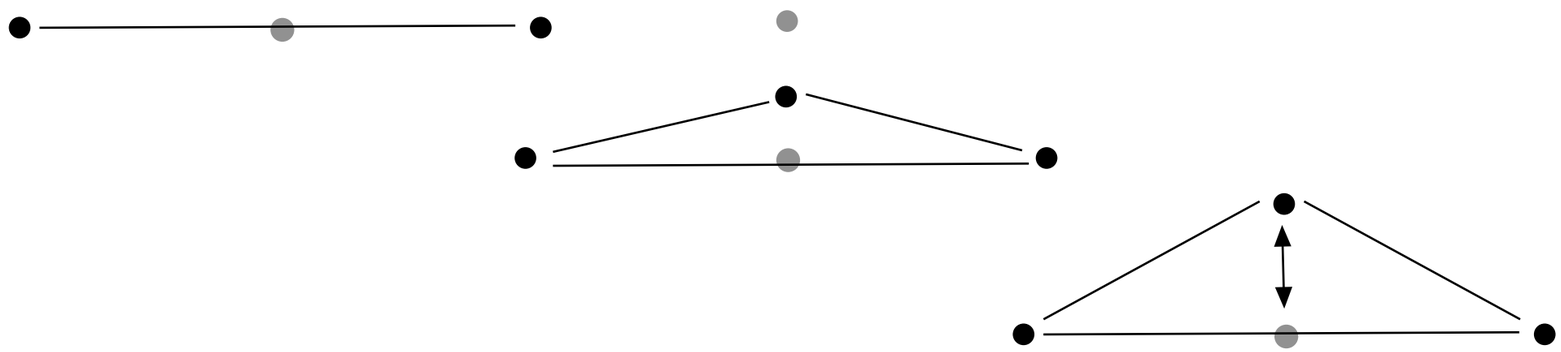
More visible than adapting resolution, but lets you work with separate patches at different resolutions



Avoid popping

Popping is solved by “morphing” between levels.

Interpolate vertices that are close to removal with the average between neighbors

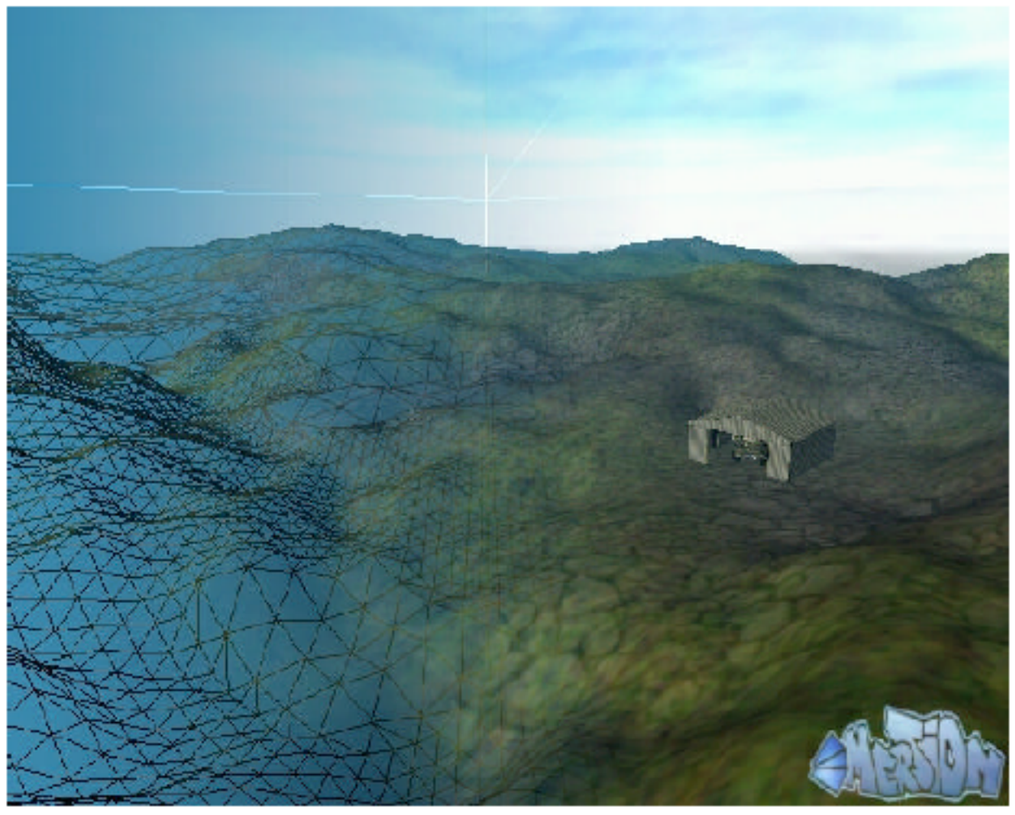


Summary: geomipmapping

- should produce polygons with roughly the same size on all distances

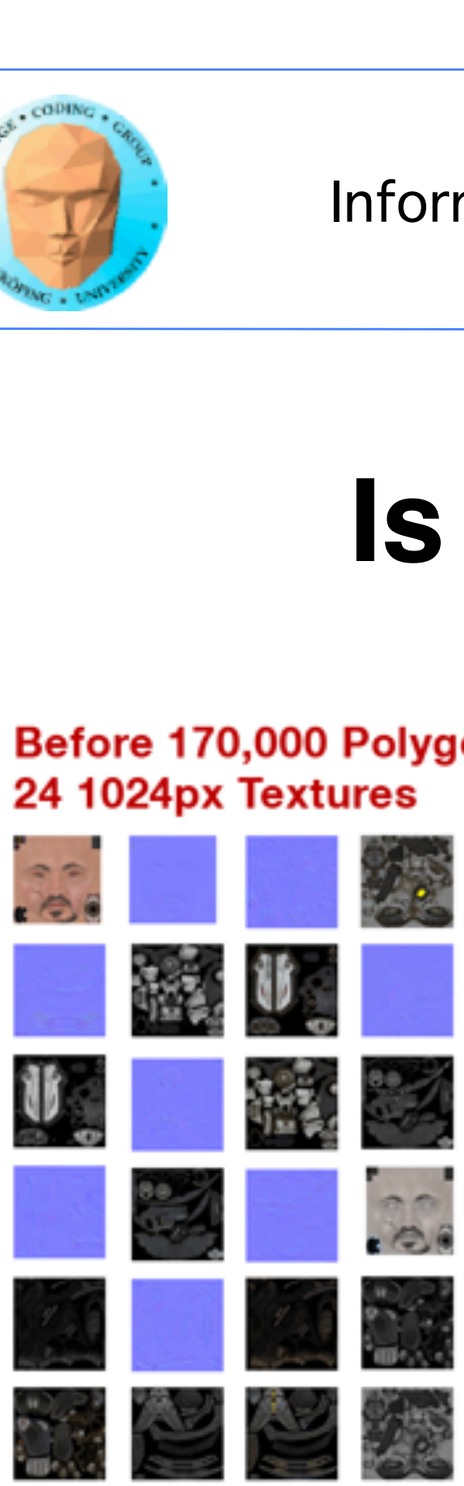
- will greatly reduce polygon count on very large terrains with large “far” distance

Geometrical mipmapping, example
(from paper by de Boer)



Is LOD important? You bet!

Before 170,000 Polygon
24 1024px Textures



After 41,000 Polygon
2 1024px Textures

SIMPLYGON

Level of detail - conclusions

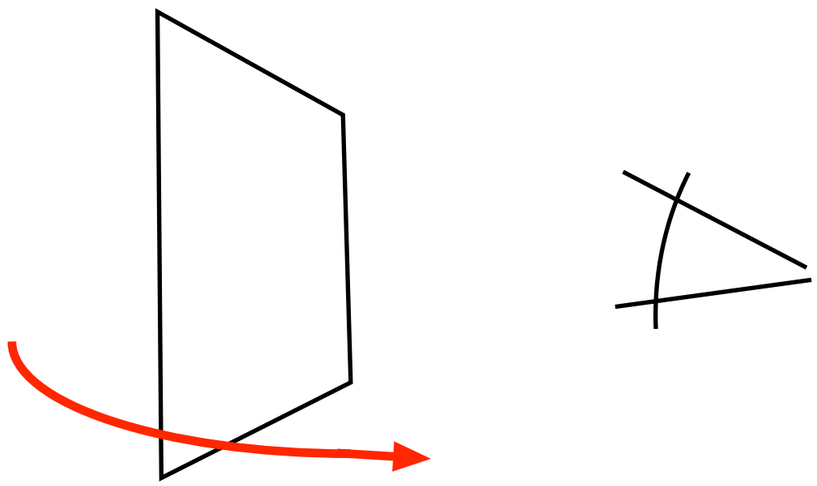
Saves vertex/polygon processing time

Can be a significant speed booster

Relatively hard to make ”perfect” result, you must fade or morph between detail levels

Billboards

A texture mapped polygon, which always faces the viewer



Billboards now and then

2D images placed on surfaces that are always facing the camera

“Sprites” in older action games

Advanced version: “impostors”

Often used for complex objects even today

Example: forests

Classic example: Doom

Big billboards. No 3D models.

Classic example: Dark Forces

Big billboards. Some 3D models.

Problem with too few views.

Billboards with transparent borders

- Billboards generally need transparency!

=> Special care is needed to avoid problems with Z-buffering!

Transparency

1) Sort by distance (Painter's algorithm)

- Perfect results
- Supports semi-transparency
- Required sorting and a separate stage

2) Discard fragments

- No semi-transparency
- Some artifacts at edges
- Very easy

3) Special case: Billboards that don't need sorting!

discard; as cheap solution to transparency

```
#version 150

out vec4 outColor;

in vec2 texCoord;
uniform sampler2D tex;

void main(void)
{
    vec4 t = texture(tex, texCoord);
    if (t.a < 0.01) discard;
    else
        outColor = t;
}
```

Special case: Billboards that don't need sorting!

additive blending

Works for billboards without detail!

Example: Snowflakes.

Blending = transparency

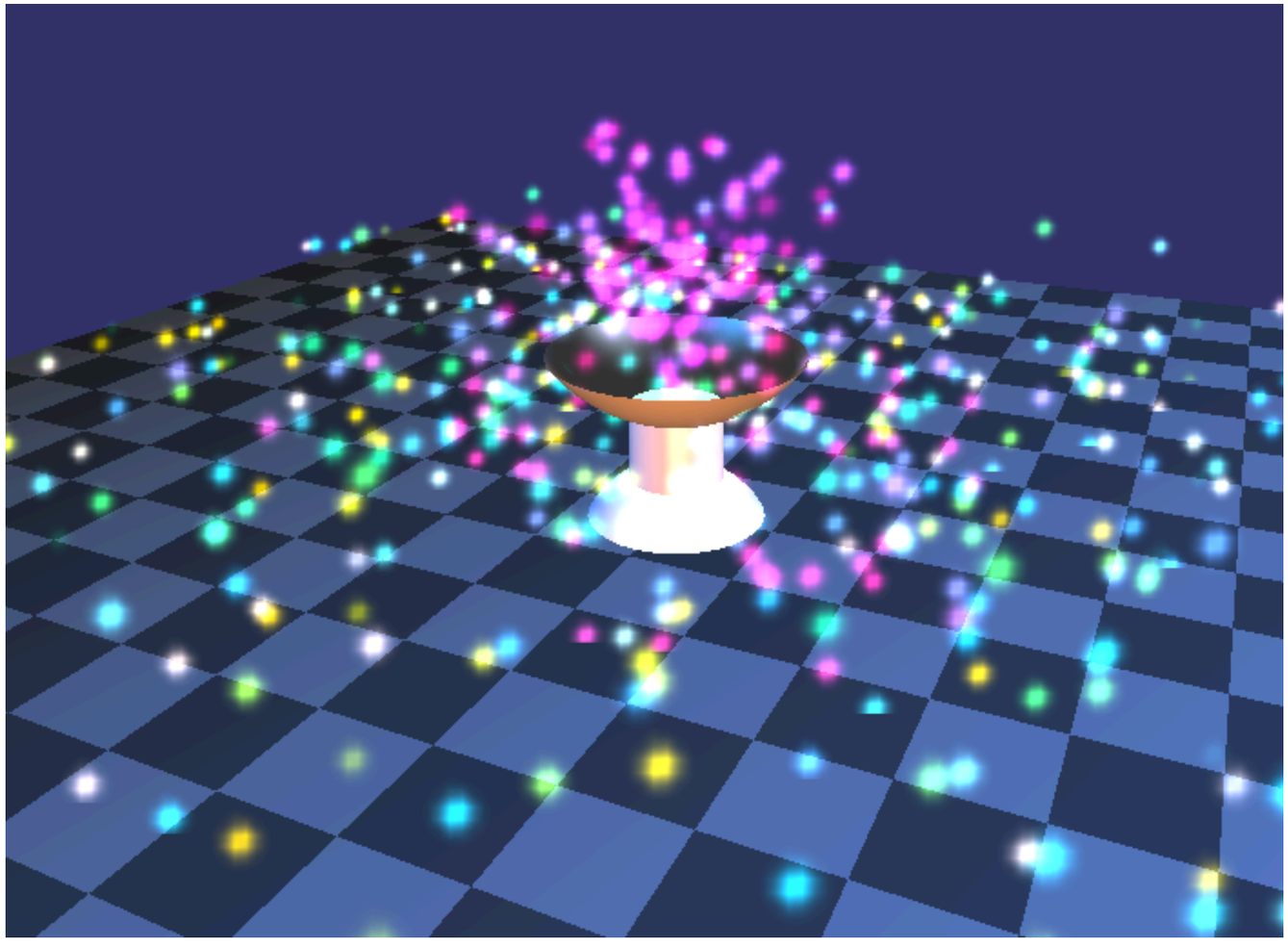
Usually

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,
      GL_ONE_MINUS_SRC_ALPHA);
```

but now it is

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

Example with additive blending



Billboard texture format

• Texture files must support transparency!

File formats:

TGA
PNG

TGA: simple format, good for demos

Libraries for PNG:
libPNG
PNGlite by Daniel Karling
LoadTexture (on my packages page)

Trick without transparency: Indicate with some unusual color.

Billboard geometry

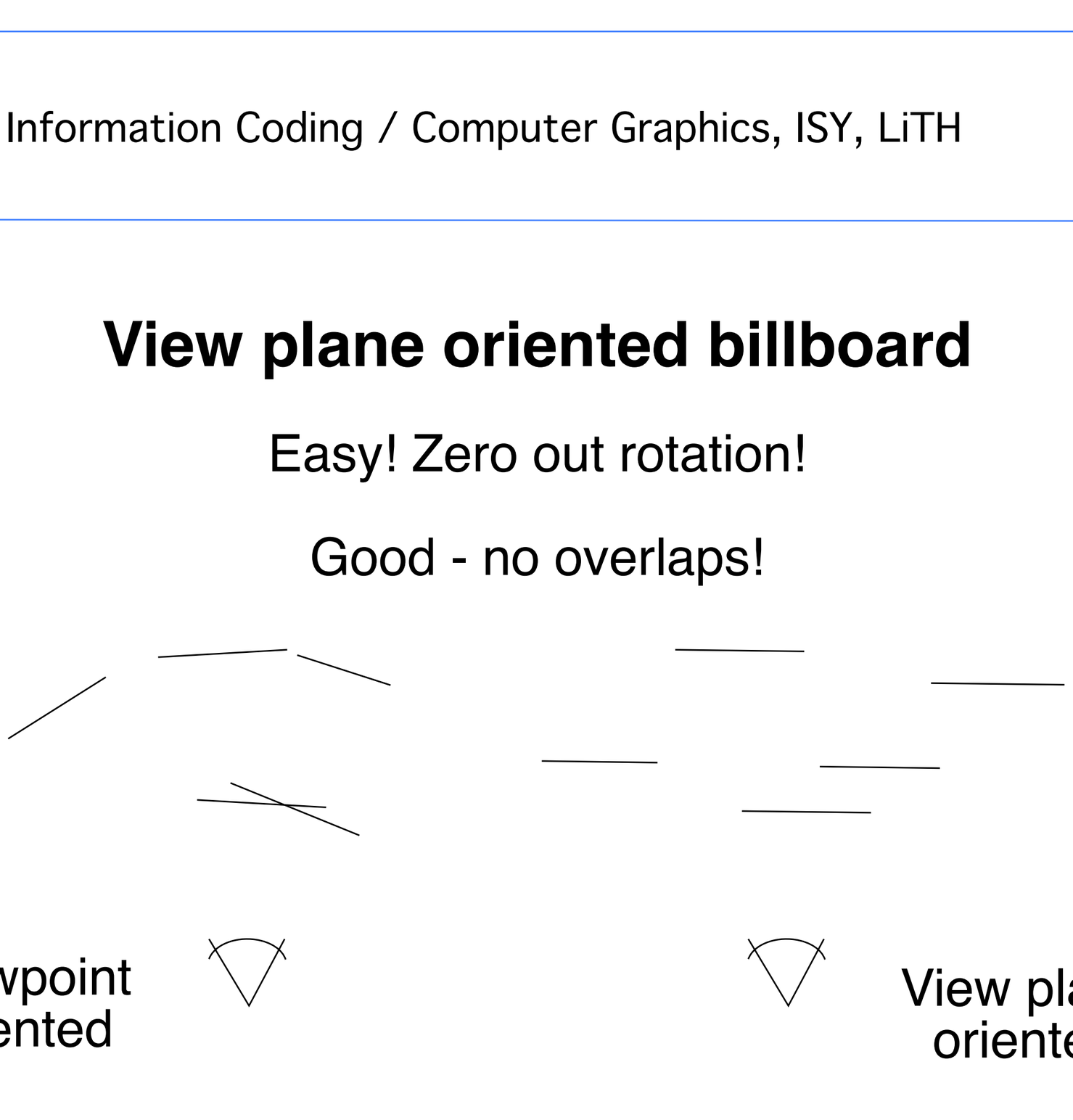
Variants:

- World oriented billboard
- Viewpoint oriented billboard (face the camera in full 3D)
- Axial (viewpoint) billboard
- View plane oriented billboard
- View plane oriented axial billboard

View plane oriented billboard

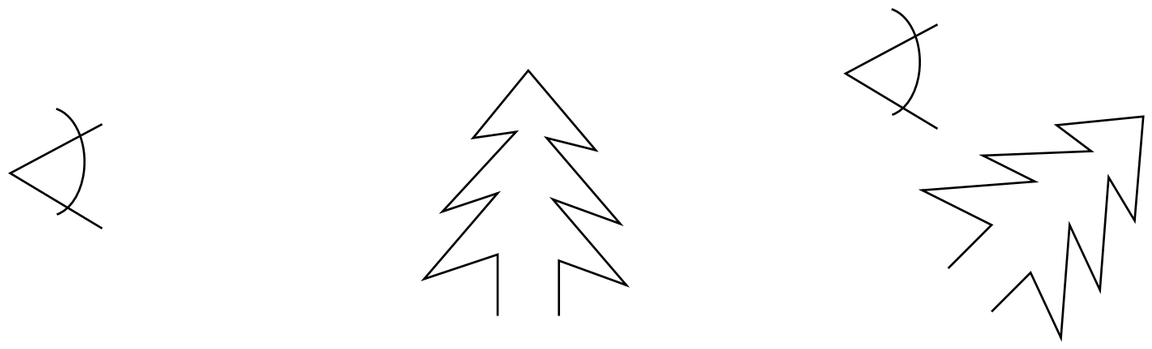
Easy! Zero out rotation!

Good - no overlaps!

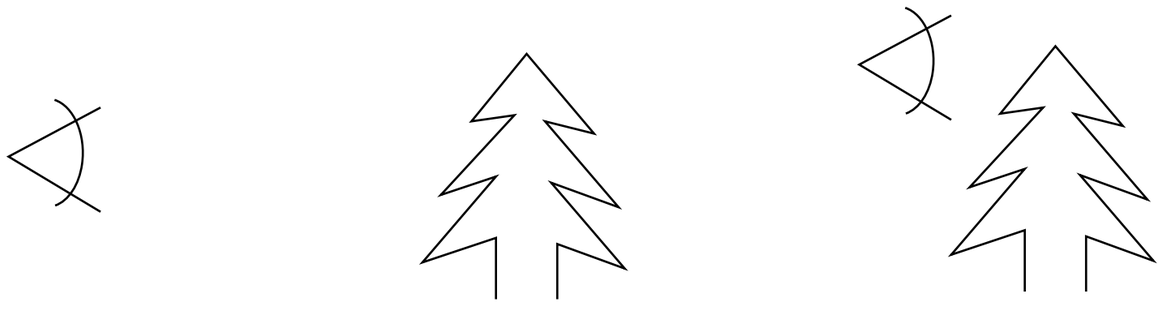


Axial billboard

Rotate around Y



Non-axial



Axial
Suitable for
trees etc

Implementing axial billboards

Project forward/eye vector on the XZ plane, find sin and cos from the normalized vector.

Same method for both, just select vector.

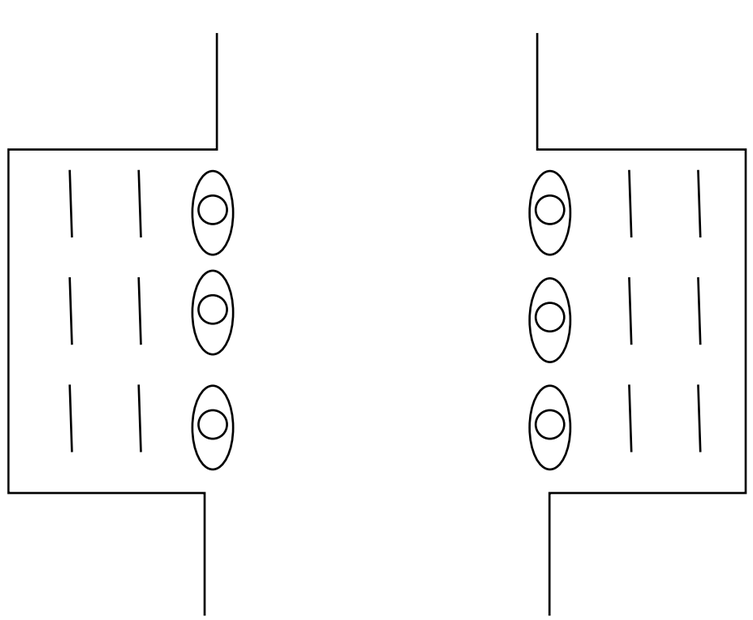
Full 3D viewpoint oriented billboard

Change of basis solution

Z vector from viewpoint, pick an up vector (usually Y axis), form basis with cross products

World oriented billboard

No camera dependent rotation



Example: Tomb Raider 2 Terracotta warriors (Temple of Xian)

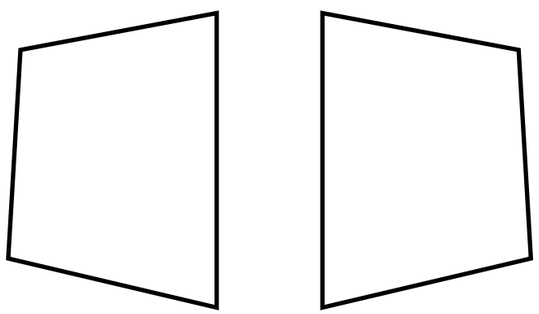
A grid of billboards

	Non-axial	Axial
Viewpoint oriented	Construct basis	Clear rotations
View plane oriented	Construct basis based on view plane	Construct axial rotation

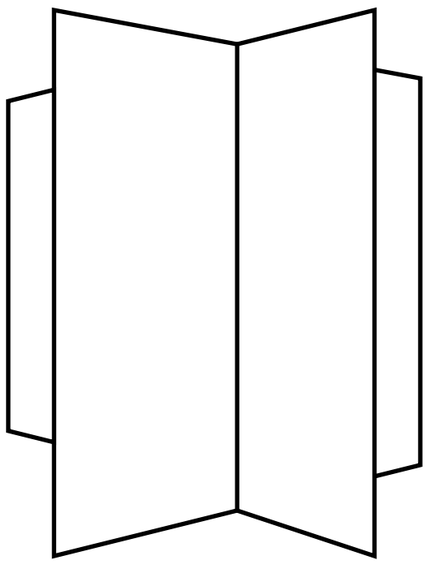
World oriented billboard not in grid

Billboard variants

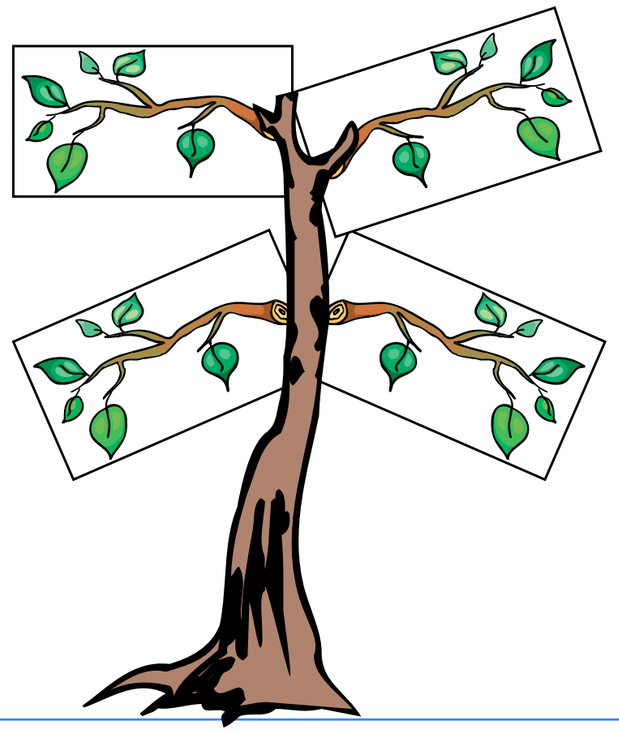
1) Always facing the camera.
One triangle or quad is enough!



2) A few polygons.
Good for world oriented billboards.



3) Multiple billboards.
Simplify complex objects om moderate distance.



Temple of Xian

Statues behind first row are billboards!

Lara's butler

The objects on the plate are billboards - some 2-poly!

And...

A more recent game makes no secret of using world oriented billboards!

Impostors

”Live” billboards

Render to texture, update sometimes

Render as other billboards

Decide when to update

Instancing

Draw a large number of the same model!

Each instance has an index, the instance number.

glDrawArraysInstanced(GL_TRIANGLES, 0, 3, 10);

draws a triangle 10 times!

gl_InstanceID tells the shader which instance we have.
Use for affecting position.

Billboard instancing demo

One single call to glDrawArraysInstanced

Position trivially affected by gl_Instance ID

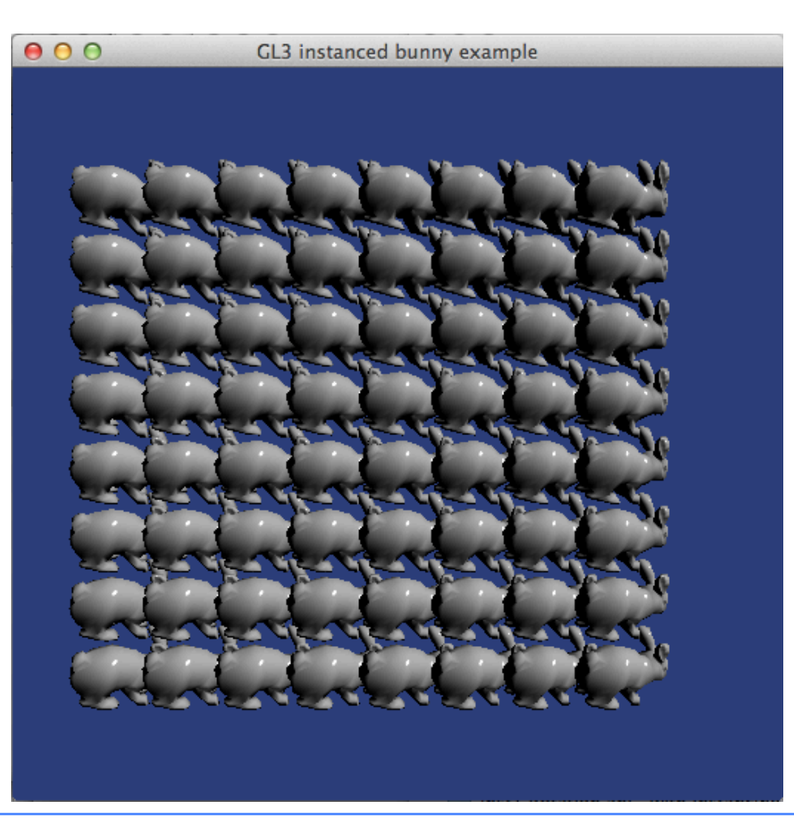
```
#version 150

in  vec3 in_Position;
uniform mat4 myMatrix;
uniform float angle;
uniform float slope;
out vec2 texCoord;

void main(void)
{
	mat4 r;
	float a = angle + gl_InstanceID * 0.5;
	float rr = 1.0 - slope * gl_InstanceID * 0.01;
	r[0] = rr*vec4(cos(a), -sin(a), 0, 0);
	r[1] = rr*vec4(sin(a), cos(a), 0, 0);
	r[2] = vec4(0, 0, 1, 0);
	r[3] = vec4(0, 0, 0, 1);
	texCoord.s = in_Position.x+0.5;
	texCoord.t = in_Position.y+0.5;
	gl_Position = r * myMatrix *  vec4(in_Position,
1.0);
}
```

Instancing complex models

Less significant; A more complex model puts enough load on the system to hide the impact of instancing.



Large worlds, conclusions

High-level VSD to limit processing to visible parts - start with frustum culling

Level-of-detail to reduce unneccessary processing of detailed models

Use billboards for extreme simplification on large distance, particle effects etc

If we can’t see the difference, use the cheaper solution!