MVD: Engine Programming

10 - Optimisation

alunthomas.evans@salle.url.edu

laSalle ENG
Universitat Ramon Llull

Optimisation

Optimising is usually something you do do later rather than earlier.

‘Getting it working is usually more important’

But before we start getting into more complex rendering stuff, some basic optimisation is probably a good idea

Profiling Tools

Profiling is the measurement of performance.

The most simple profiling is measuring average ms/frame and outputing the value to the console every *x* frames

CPU profiling

Visual Studio has a very good in depth profiler for CPU usage

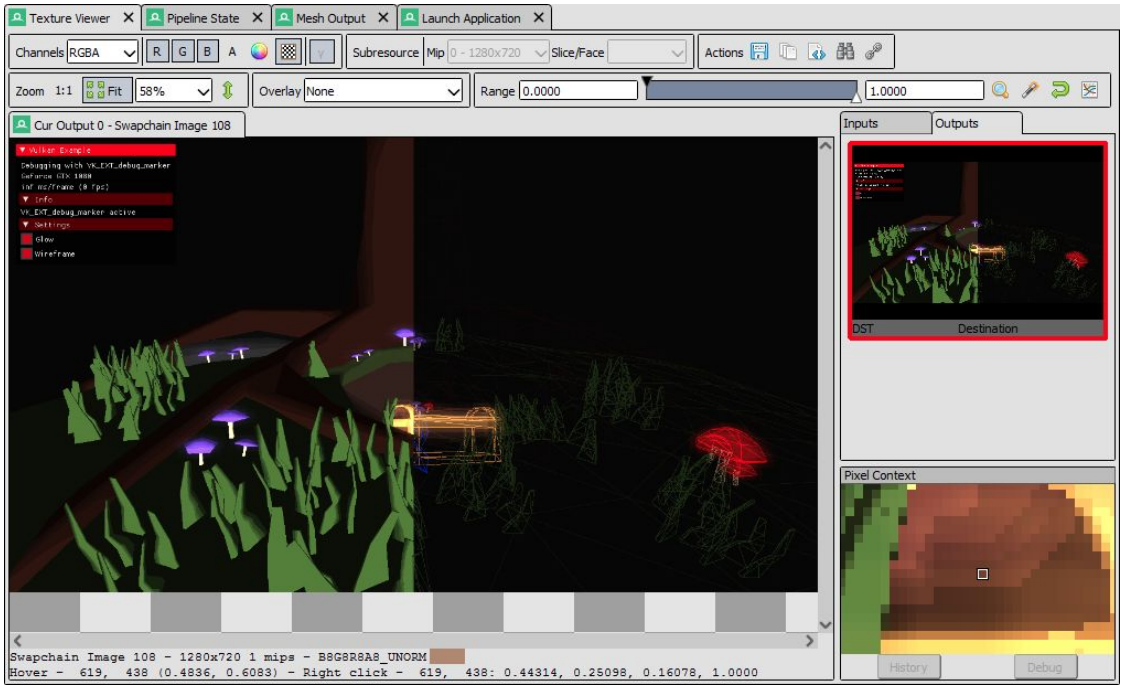
It is very good at finding mistakes which are simple to fix e.g. std::vector.push_back() somewhere in loop

GPU profiling

Renderdoc is an excellent open source GPU profiler and debugging tool.

https://renderdoc.org/

You ‘attach’ it to an OpenGL process, and it will measure the the impact of all GL calls, show texture output etc.



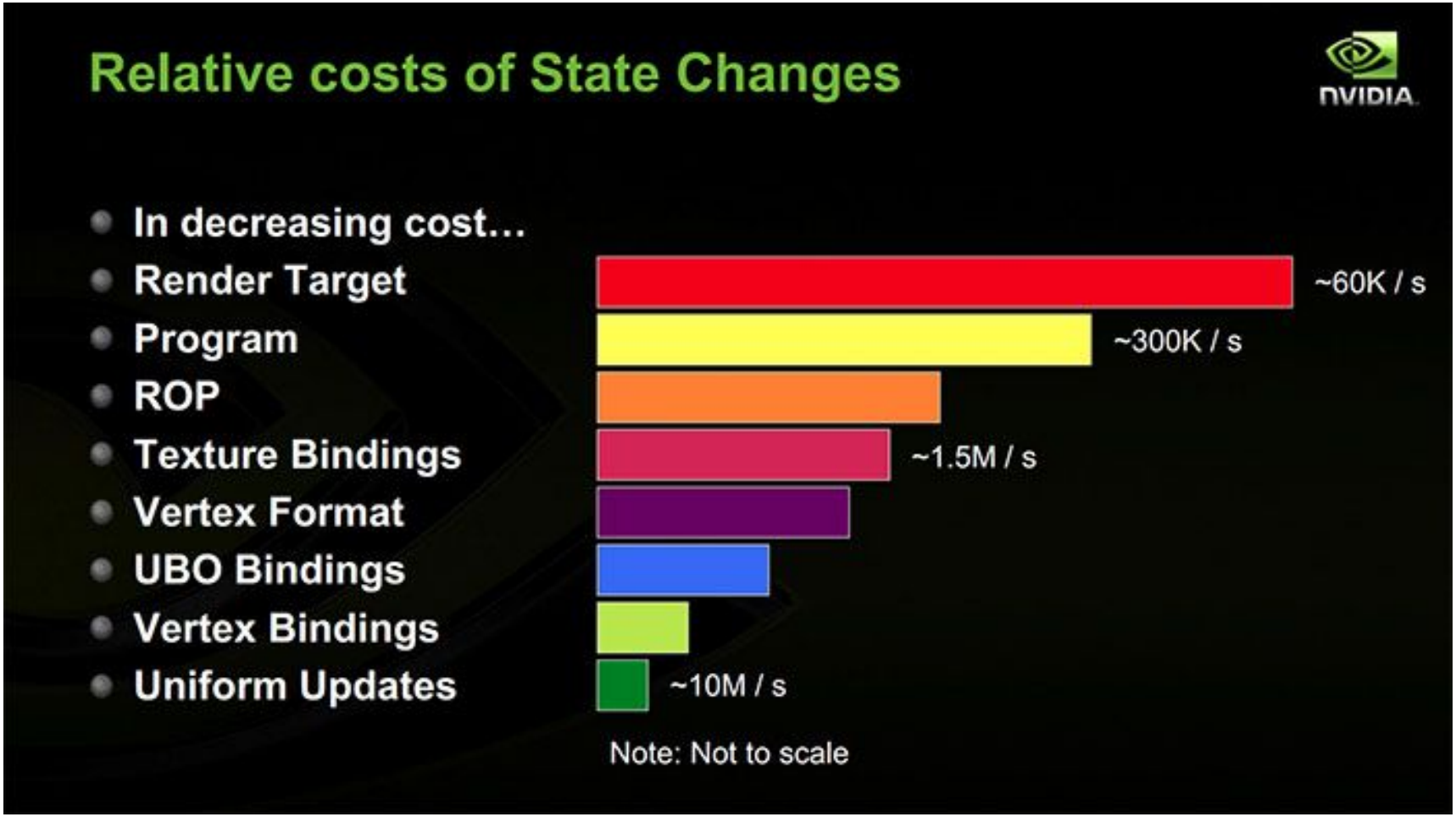
Three Basic (Essential) Optimisations!

1) State change optimisations

2) Shader class optimisation

3) Frustum culling

Avoiding changing OpenGL state



Minimising OpenGL state change

```
foreach shader {
  set shader state (e.g. shader, tessellation…)
  foreach material {
     set material state (e.g. textures, uniforms)
     foreach object/geometry {
        set object/geometry state (e.g. vertex/index buffers, matrices)
        draw calls
     }
  }
}
```

Classic approach: create ‘RenderInstances’ or ‘Renderlist’ of Meshes, for each shader/material

Sort our array of mesh Components

Our current engine draws all Mesh Components in sequence, we don't want to lose this by creating render instances.

But can solve problem if we sort Mesh component array by two parameters:

1) Shader program id (stored in material)

2) Material id (stored in Mesh component)

Sorting algorithms: quicksort

std::sort

Uses quicksort algorithm to sort an array:

```
std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

std::sort(s.begin(), s.end());

//output
//0 1 2 3 4 5 6 7 8 9
```

std::sort objects

Need to provide function by which std::sort will evaluate. e.g. to sort materials by the id of their shader:

```
std::sort( materials_.begin(),
           materials_.end(),
           [](const Material& a, const Material& b) {
                   return a.shader_id < b.shader_id;
           });
```

lambda expression - self contained function that can be passed as a variable

In our engine - tracking indices after sort

The big problem we have is that our entire engine relies on tracking items by index - (not pointers)

When we sort, we change indices, so must update references!

Can use std::map<int, int> to map old indices to new indices, then update

In our engine

Sort MeshComponents by shader, then material, in lateInit function

Before rendering mesh component, check if we need to change shader, or change material. Then render meshes:

```
auto& mesh_components = ECS.getAllComponents<Mesh>();
for (auto &mesh : mesh_components) {
    checkShaderAndMaterial(mesh);
    renderMeshComponent_(mesh);
}
```

Guaranteed to have most efficient rendering order

checkShaderAndMaterial(Mesh& mesh)

if (*current shader* != *current material's shader*)

change current shader to that of material

if (*current material* != *current mesh's material*)

update material uniforms

```
void GraphicsSystem::checkShaderAndMaterial(Mesh& mesh) {
    //get shader id from material. if same, don't change
    if (!shader_ || shader_->program != materials_[mesh.material].shader_id) {
        useShader(materials_[mesh.material].shader_id);
    }
    //set material uniforms if required
    if (current_material_ != mesh.material) {
        current_material_ = mesh.material;
        setMaterialUniforms();
    }
}
```

Task

Fill in the functions to sort our mesh array first by shader, then by material

A better shader

A better shader

Take this code:

```
//transform uniforms
GLint u_mvp = glGetUniformLocation(current_program_, "u_mvp");
if (u_mvp != -1) glUniformMatrix4fv(u_mvp, 1, GL_FALSE, mvp_matrix.m);
```

it is executed every frame.

How can we optimise this code?

Uniform location

glUniformLocation returns the **same value** until shader is recompiled - but then may be **different**

...and comparing by string is the the most expensive comparison you can possibly do (don't do it, unless you can avoid it)

So let's call glUniformLocation once for each uniform, and cache the result.

How can we cache the uniform locations?

Uniform Location caching

Option A:

Use a std::unordered_map e.g. in shader init:

```
std::unordered_map<std::string, GLint> uniforms;
uniforms["u_mvp"] = glGetUniformLocation(program, "u_mvp");
```

then in render loop:

```
//in render loop
glUniformMatrix4fv(shader.uniforms["u_mvp"], 1, GL_FALSE, mvp_matrix.m);
```

Uniform caching without string comparison

std::unordered_map<std::string, int> is **slow** as it used a string for the key

std::unordered_map<int, int> would be faster. So let's make an enum of all possible uniform variables.

We will store these values in our uniform map

```
enum UniformID {
    U_VP,
    U_MVP,
    U_MODEL,
    U_NORMAL_MATRIX,
    U_CAM_POS,
    U_COLOR,
    U_COLOR_MOD,
    U_AMBIENT,
    U_DIFFUSE,
    U_SPECULAR,
    U_SPECULAR_GLOSS,
    U_USE_DIFFUSE_MAP,
    U_DIFFUSE_MAP,
    U_SKYBOX,
    U_USE_REFLECTION_MAP,
    U_NUM_LIGHTS,
    UNIFORMS_COUNT
};
```

Uniform caching without string comparison

Now to store uniforms, we don't even need to use std::unordered_map - given that the enums are linear, a std::vector is fine!

```
//stores, for each uniform enum, it's location
std::vector<GLuint> uniform_locations_;
```

Hard coding string->enum

However, we *do* have to store somewhere the enum strings, but we hard code them into our engine, and map them to the enum values:

When initialising the shader, we use this map to store the value returned from glGetUniformLocation in our std::vector storage

```
const std::unordered_map<std::string, UniformID> uniform_string2id_ = {
    { "u_vp", U_VP },
    { "u_mvp", U_MVP },
    { "u_model", U_MODEL },
    { "u_normal_matrix", U_NORMAL_MATRIX },
    { "u_cam_pos", U_CAM_POS },
    { "u_color", U_COLOR },
    { "u_color_mod", U_COLOR_MOD },
    { "u_ambient", U_AMBIENT },
    { "u_diffuse", U_DIFFUSE },
    { "u_specular", U_SPECULAR },
    { "u_specular_gloss", U_SPECULAR_GLOSS },
    { "u_use_diffuse_map", U_USE_DIFFUSE_MAP },
    { "u_diffuse_map", U_DIFFUSE_MAP },
    { "u_skybox", U_SKYBOX },
    { "u_use_reflection_map", U_USE_REFLECTION_MAP },
    { "u_num_lights", U_NUM_LIGHTS }
};
```

Abstract functions to set uniforms

```
//Returns location of uniform with given enum
GLuint Shader::getUniformLocation(UniformID name) {
    return uniform_locations_[name];
}


//set vec3 array
bool Shader::setUniform(UniformID id, const lm::vec3& data) {
    GLuint loc = getUniformLocation(id);
    if (loc != -1) {
        glUniform3fv(loc, 1, data.value_);
        return true;
    }
    return false;
}
```

Shader optimisations in Graphics System

Previously we stored our shaders in the graphics system as

```
std::unordered_map<std::string, Shader*> shaders_; //name, pointer
```

now we do it as using openGL id:

```
std::unordered_map<GLint, Shader*> shaders_; //compiled id, pointer
```

because it enables us to map the compiled id to the pointer to the shader = greater flexibility

Task

Change all instances of glGetUniform for the the new setUniform function

As you do this, find out the answer as to why we can't easily do it for the 'light' uniforms

Frustum culling

cull verb

\ˈkəl \

culled; culling; culls

Definition of *cull* (Entry 1 of 2)

transitive verb

1 : to select from a group : CHOOSE

// *culled* the best passages from the poet's work

// Damaged fruits are *culled* before the produce is shipped.

2 : to reduce or control the size of (something, such as a herd) by removal (as by hunting) of especially weaker animals

also : to hunt or kill (animals) as a means of population control

// The town issued hunting licenses in order to *cull* the deer population.

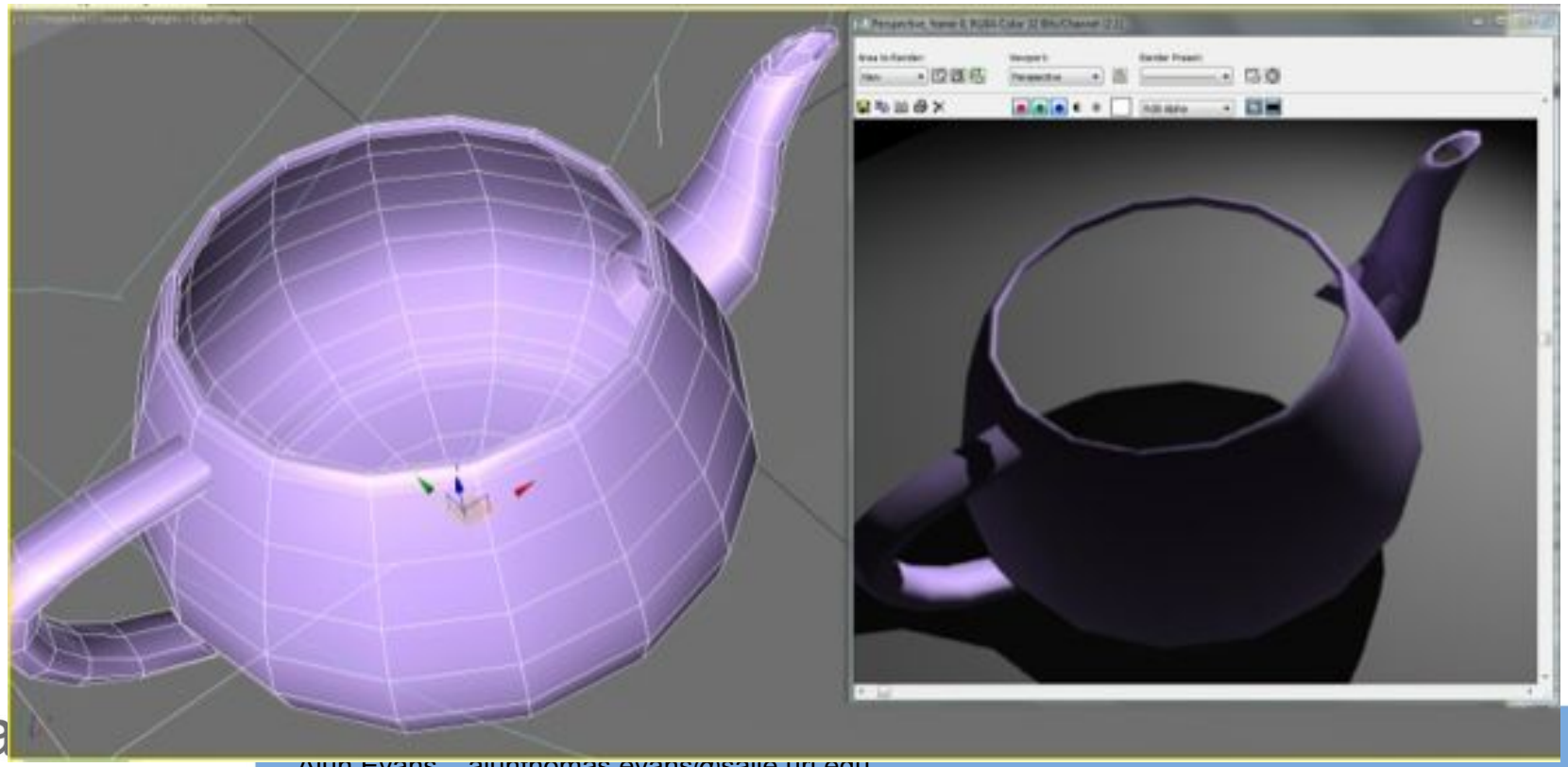
Culling in graphics

"Not drawing things the user can't see"

Face culling in OpenGL

Only drawing one side of each triangle

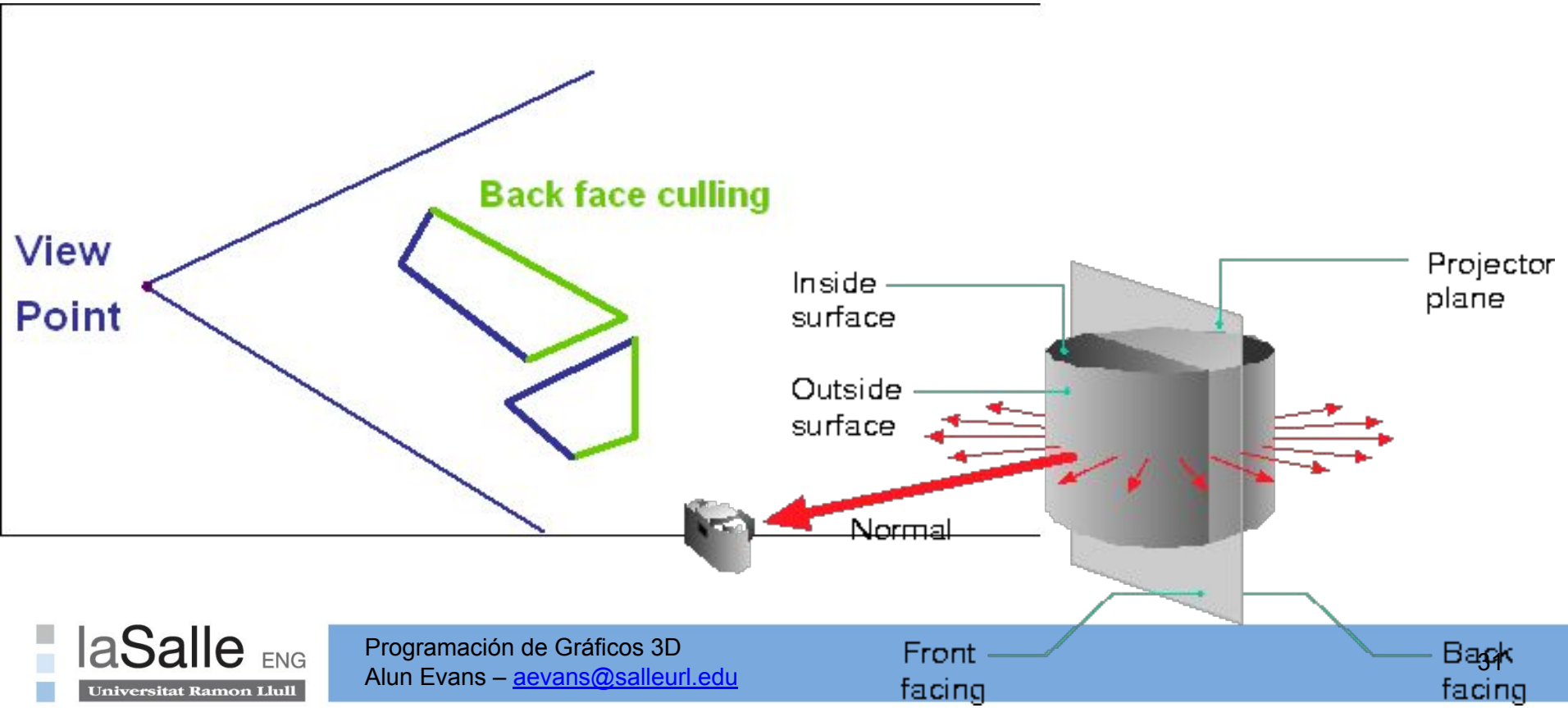
```
glEnable(GL_CULL_FACE); //enable culling
glCullFace(GL_BACK); //which face to cull
```



Backface culling

OpenGL face culling only removes one side of triangle. It does not test if face is facing the camera or not.

Use dot product of Normal with Camera Vector to



Backface culling

In reality is not that useful, because

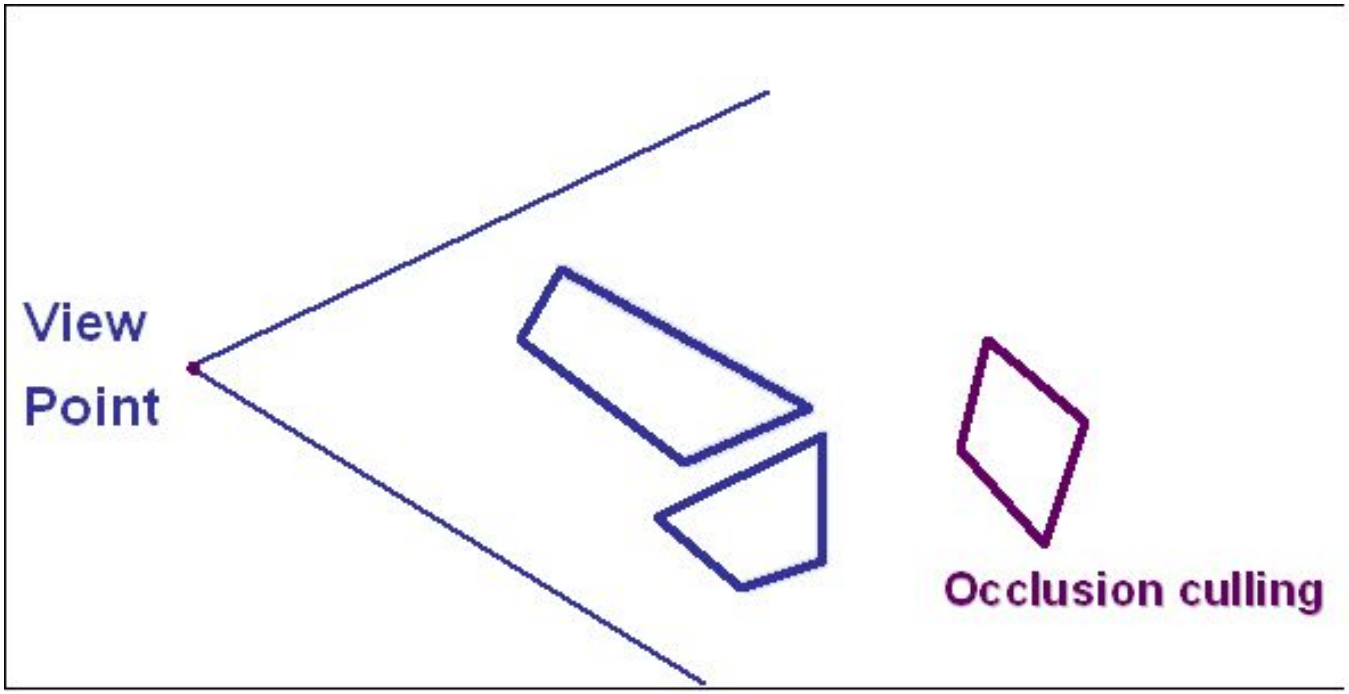
1) it's quite expensive
   - Have to edit index buffer every frame for every mesh
   - have to write a geometry shader to remove geometry from pipeline

2) OpenGL culling, while not 'correct', is frequently good enough

3) The depth test prevents needless calculation of hidden surfaces anyway

Occlusion culling

Cull **objects** based on whether the can be seen or not



How do we implement this?

Occlusion culling

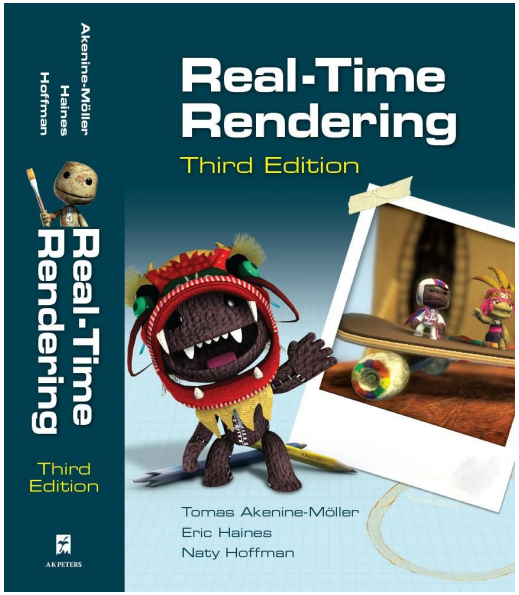
Quite a complicated topic.

But it is very much ‘solved’.

See Real Time Rendering (Akenine-Moller)

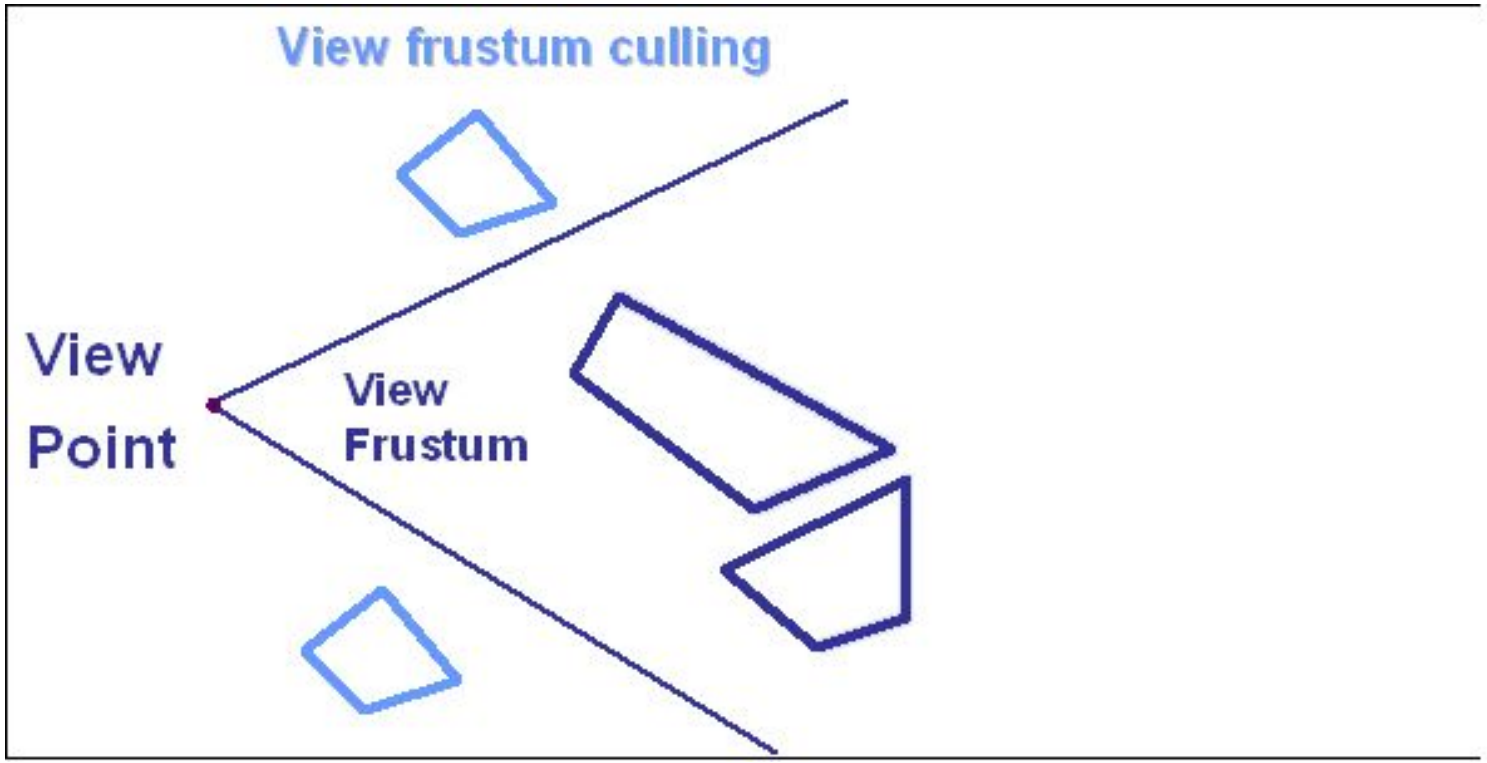
and

http://www.gamasutra.com/view/feature/131801/occlusion_culling_algorithms.php

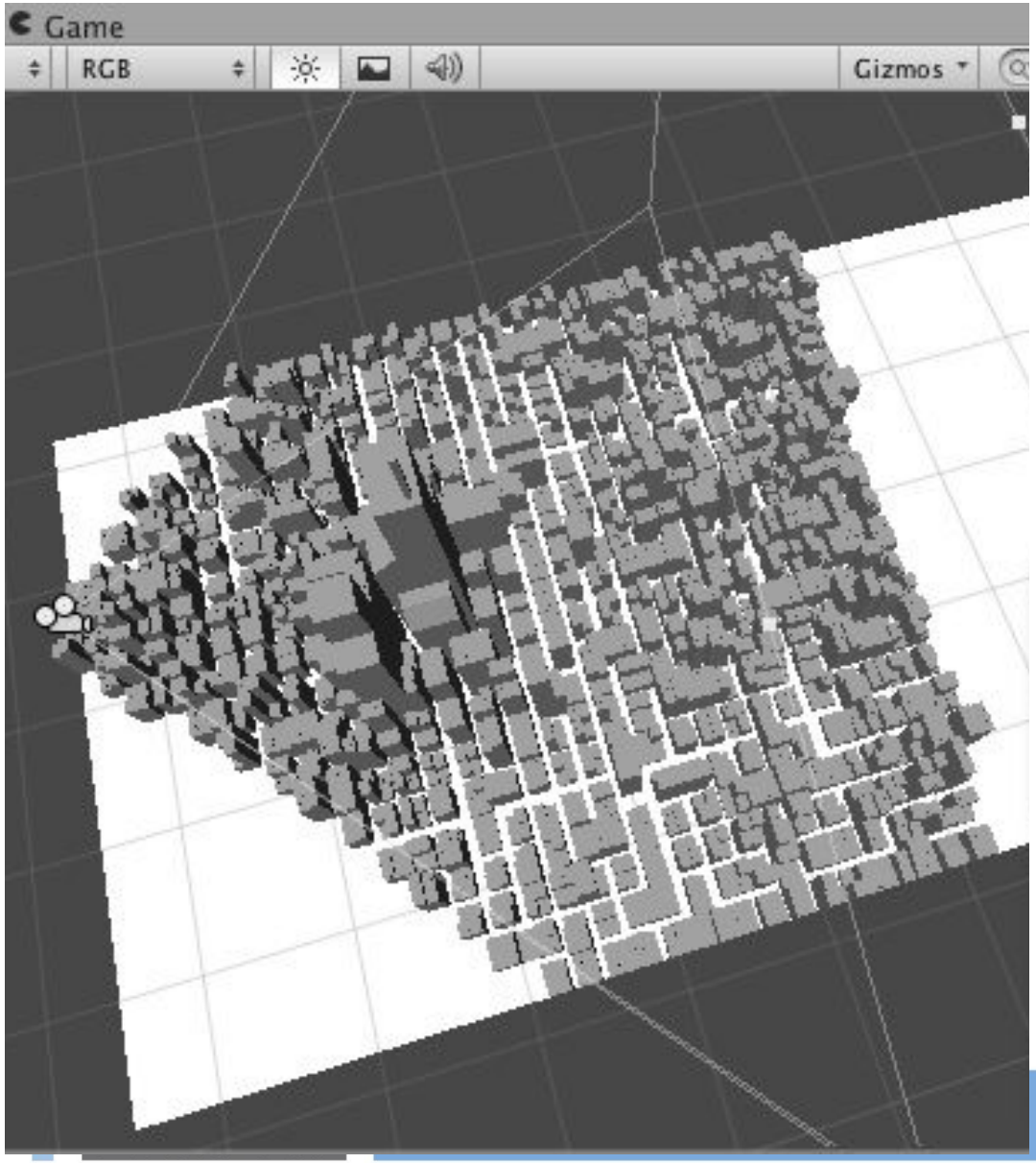


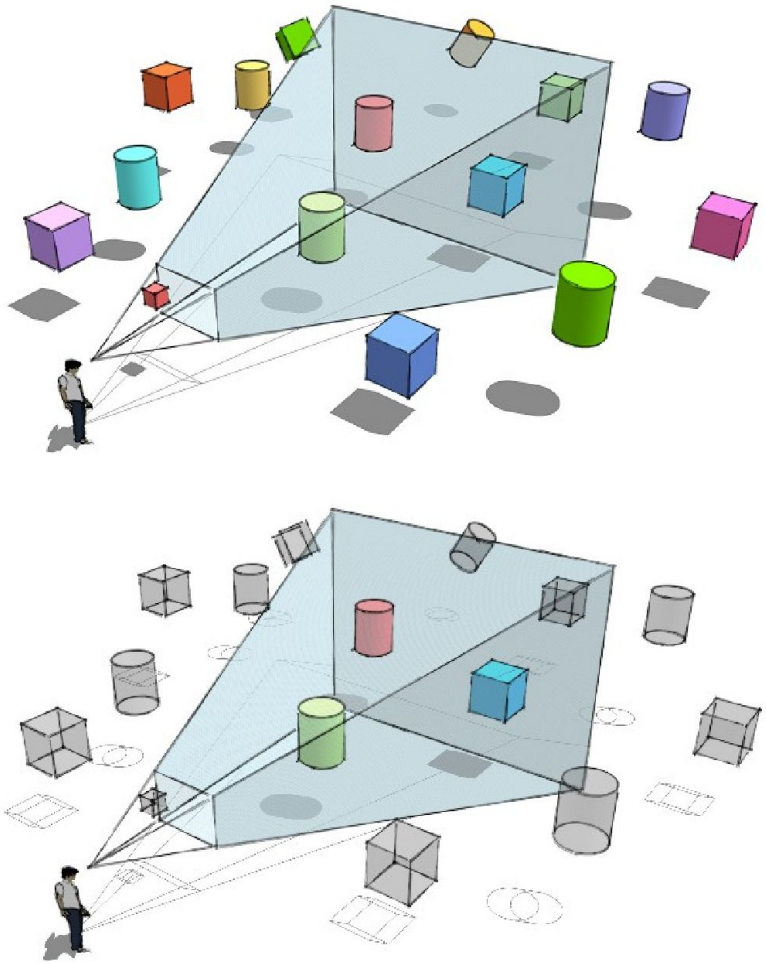
Frustum culling

Cull **objects** based on whether they are in frustum or not



View Frustum Culling





View Frustum Culling

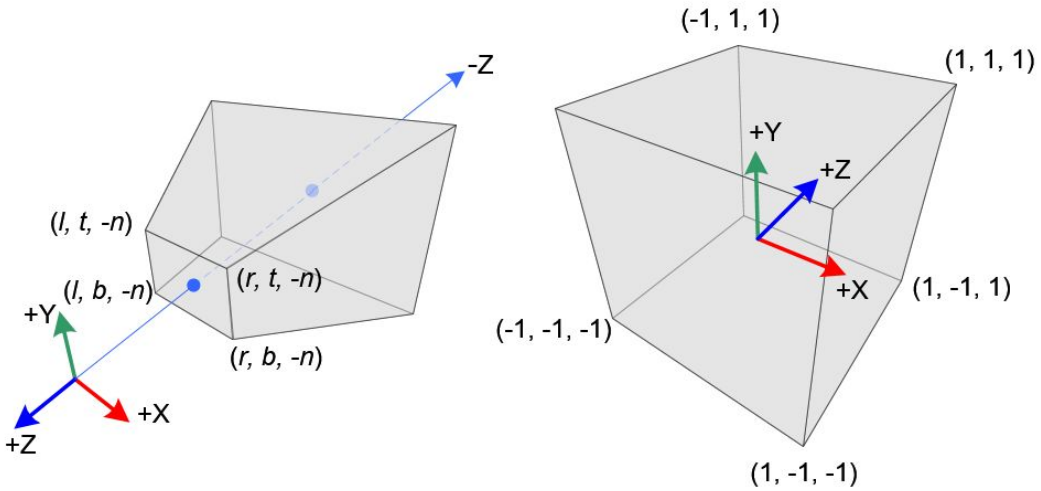
Do not draw anything that is not inside the camera frustum.

*How do we test if a point, **P**, in inside the frustum or not?*

View Frustum Culling

Do not draw anything that is not inside the camera frustum.

*How do we test if a point, **P**, in inside the frustum or not?*



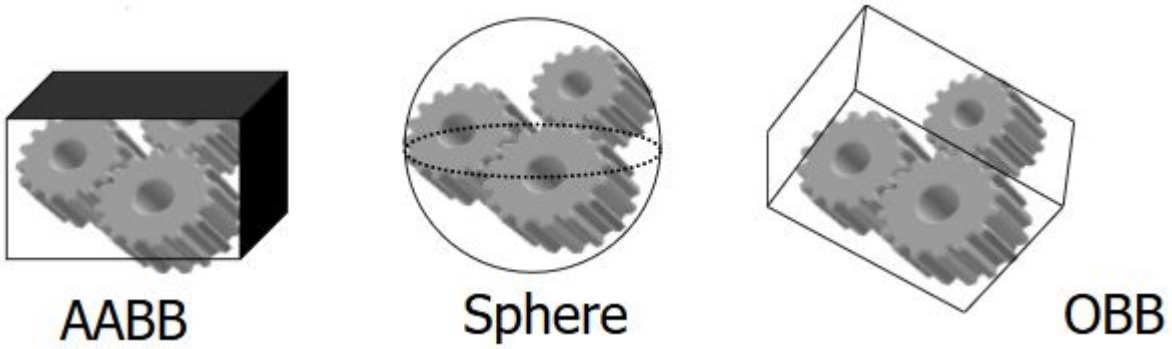
Simple: if its coordinates in clip space are -1 < P < +1

View Frustum Culling

Testing all points of mesh is obviously not optimal.

So we use a **bounding volume.**

Why not use a collider?

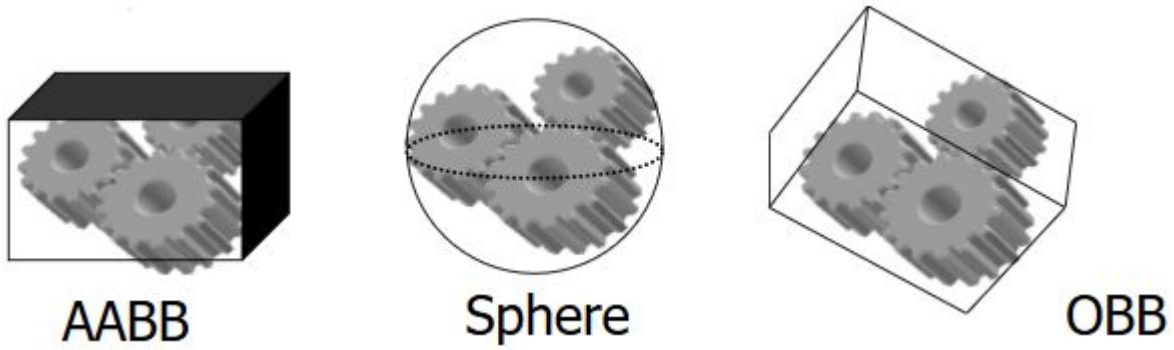


View Frustum Culling

Testing all points of mesh is obviously not optimal.

So we use a **bounding volume.**

Why not use a collider? We could, but we can't guarantee that every object will a) have one, and b) it be tight



AABB creation

Iterate all vertices of mesh, store min max in each axis, then calculate center and halfwidth

```
struct AABB {
    lm::vec3 center;
    lm::vec3 half_width;
};
```

AABB in frustum

Using center and halfwidth, calculate 8 points of AABB.

Multiply all point by ModelViewProjection matrix

Test each point to see if value is between -1 and +1 in each axis.

Tasks

Code create AABB