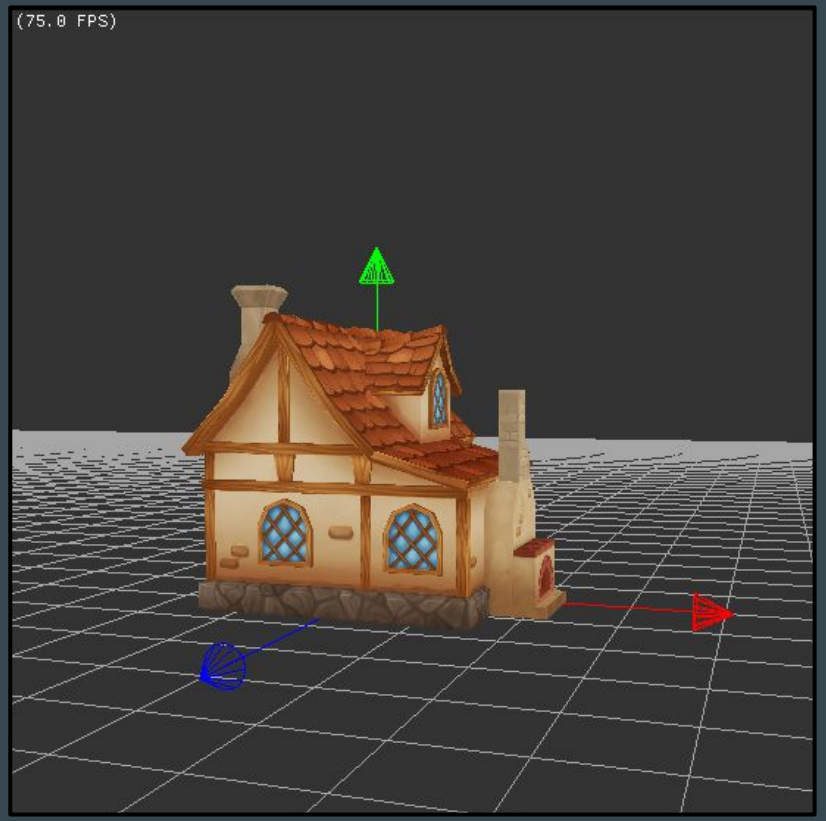
Geometry loading

...

Carlos Fuentes

Goal



Geometry pipeline

- The **geometry pipeline** is a process used to modify model data from the origin (3DS Max or Maya) into our game engine.
- Usually has following stages:
  a. Export to an format for interchange: **FBX, Collada, glTF**, etc.
  b. Import from generic format to our **own format**.
  c. Loading into RAM or Video memory through **VBO**.

Assimp

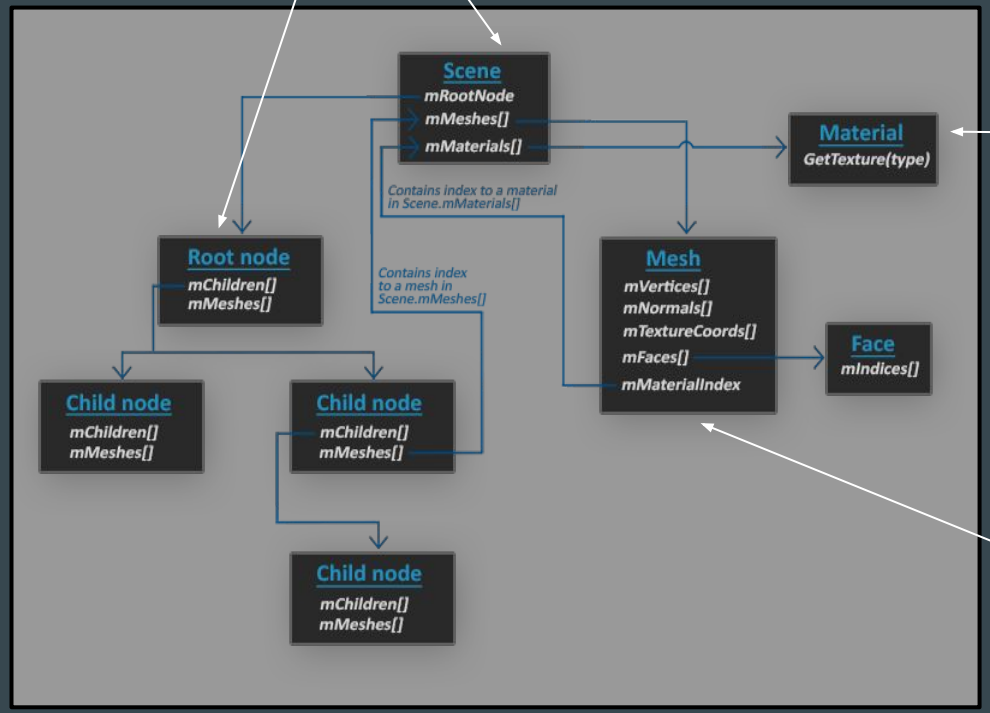
- We are going to use assimp, a library able to load multiple formats for models and animations. Supports formats like:
  - **FBX** : Autodesk binary format
  - **Collada**: Public Text format
  - **glTF**: is becoming a new standard. Binary forma.
- For our project we will use it to load models and generate **our own format**
- But, for the moment, we will use it to load models to memory using a **VBO**.

Assimp download and installation

- Download assimp last version and build for **x64** platform and **Release**
- Installation:
  a. Copy assimp include directory into an assimp folder (inside our project folder)
  b. Copy assimp-v141-mt.lib into assimp folder
  c. Copy assimp-v141-mt.dll into Game folder.
  d. In your project properties inside **Visual Studio**:
    - Update additional include directories, additional library directories
    - Add assimp-v141-mt.lib as input library

Assimp classes

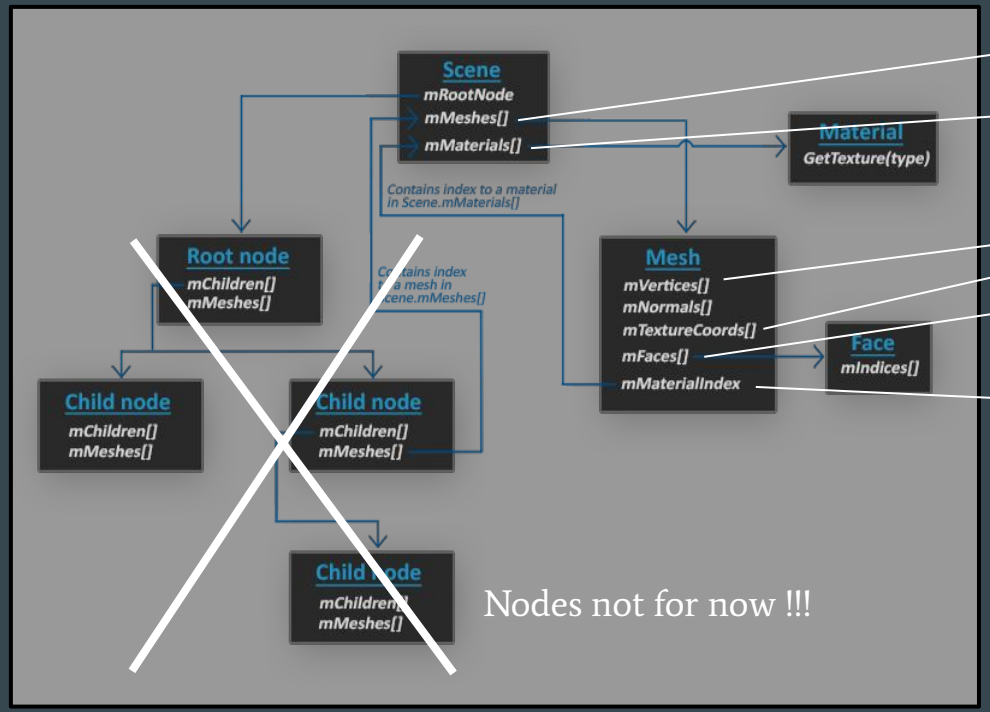
```
#include <assimp/scene.h>
```




```
#include <assimp/material.h>
```

```
#include <assimp/mesh.h>
```

Assimp classes



Model
Mesh[]
Texture[]

Mesh
VBO
EBO
VAO
material_index
num_vertices
num_indices

Model class: Load

- Create a new class **Model** analogous to assimp **aiScene** class, and load all meshes and textures.

```
void Model::Load(const char* file_name)
{
    const aiScene* scene = aiImportFile(file_name, aiProcessPreset_TargetRealtime_MaxQuality );
    if(scene)
    {
        // TODO: LoadTextures(scene->mMaterials, scene->mNumMaterials);
        // TODO: LoadMeshes(scene->mMeshes, scene->mNumMeshes);
    }
    else
    {
        LOG("Error loading %s: %s", file,aiGetErrorString());
    }
}
```

Model class: Load Material

- Materials contains shading information of a mesh.
- For now we are only interested in retrieving diffuse texture.
- Load example:

```
void Model::LoadMaterials(const aiScene* scene)
{
    aiString file;

    materials.reserve(scene->mNumMaterials);

    for(unsigned i=0; i< scene->mNumMaterials; ++i)
    {
        if(scene->mMaterials[i]->GetTexture(aiTextureType_DIFFUSE, 0, &file) == AI_SUCCESS)
        {
            materials.push_back(App->textures->Load(file.data));
        }
    }
}
```

Mesh class: Load

- Each mesh must create a VBO with vertex positions and texture coordinates from assimp class **aiMesh**:
    a. **mVertices** → vertex positions
    b. **mTextureCoords** → texture coordinates
    c. **mNumVertices** → number of vertices (valid por positions, texture coordinates, etc.)

Mesh class: VBO creation

- We were using glBufferData function to initialize VBO, passing a custom buffer with whole data: positions followed by texture coordinates.
- Assimp provides:
    a. Array of **positions**
    b. Array of **texture coordinate sets**
        i. Each coordinate set contains texture coordinates for whole mesh
        ii. We are going to support for now 1 texture coordinate set so, load always set 0.

Mesh class: VBO creation

- Is it possible to call glBufferData passing nullptr as buffer and, later, update data using:
  - glBufferSubData updates a range of VBO data with given buffer
  - glMapBuffer returns a pointer to VBO data so you can write/read from it.
  - glMapBufferRange the same as glMapBuffer but for a range of data.
  - glUnmapBuffer must be called when you are done with glMapBuffer or glMapBufferRange pointer.

Mesh class: VBO creation

- glBufferSubData and glMapBufferRange example

```
void Mesh::LoadVBO(const aiMesh* mesh)
{
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);

    unsigned vertex_size = (sizeof(float)*3+sizeof(float)*2);
    unsigned buffer_size = vertex_size*mesh->mNumVertices;
    glBufferData(GL_ARRAY_BUFFER, buffer_size, nullptr, GL_STATIC_DRAW);

    unsigned position_size = sizeof(float)*3*mesh->mNumVertices;
    glBufferSubData(GL_ARRAY_BUFFER, 0, position_size, mesh->mVertices);

    unsigned uv_offset = position_size;
    unsigned uv_size   = sizeof(float)*2*mesh->mNumVertices;
    float2* uvs = (float2*)(glMapBufferRange(GL_ARRAY_BUFFER, uv_offset, uv_size, GL_MAP_WRITE_BIT));

    for(unsigned i=0; i< mesh->mNumVertices; ++i)
    {
        uvs[i] = float2(mesh->mTextureCoords[0][i].x, mesh->mTextureCoords[0][i].y);
    }
    glUnmapBuffer(GL_ARRAY_BUFFER);
    num_vertices = mesh->mNumVertices;
 }
```

Mesh class: VBO creation

- We have been storing vertex attributes (positions, texture coordinates) as **separated** (or spread out) arrays
- There is another option, store vertex attributes as **interleaved** arrays

```
float data[] = {
      -1.0f, -1.0f, 0.0f,   // ← v0 pos
       1.0f, -1.0f, 0.0f,   // ← v1 pos
       0.0f,  1.0f, 0.0f    // ← v2 pos

       0.0f, 0.0f,          // ← v0 texcoord
       1.0f, 0.0f,          // ← v1 texcoord
       0.5f, 1.0f           // ← v2 texcoord
     };
```

```
float data[] = {
      -1.0f, -1.0f, 0.0f,   // ← v0 pos
       0.0f, 0.0f,          // ← v0 texcoord

       1.0f, -1.0f, 0.0f,   // ← v1 pos
       1.0f, 0.0f,          // ← v1 texcoord

       0.0f,  1.0f, 0.0f    // ← v2 pos
       0.5f, 1.0f           // ← v2 texcoord
     };
```

Mesh class: Separated vs Interleaved Arrays

- **Interleaved arrays**→ continuous read of whole vertex → better memory access for drawing (vertex shader) → good for **static** meshes
- **Separated arrays** → continuous write of single attribute → good for **dynamic** meshes

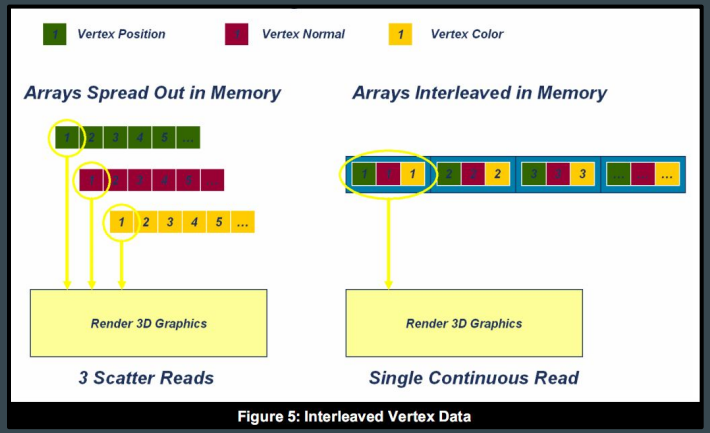


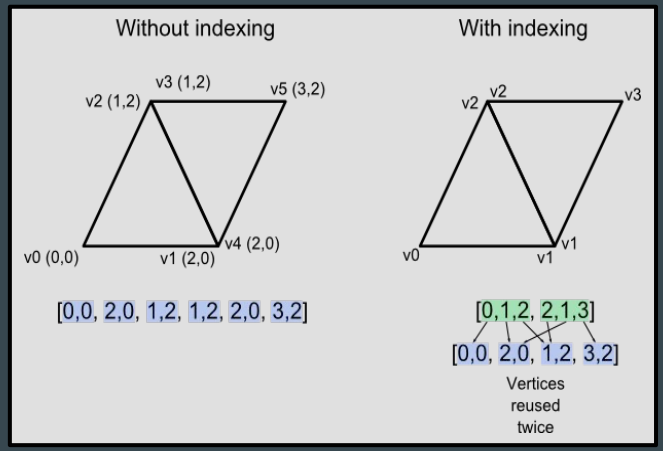
Figure 5: Interleaved Vertex Data

Mesh class: Interleaved arrays attributes

- Attributes are described using void **glVertexAttribPointer** function. For **interleaved** arrays we must use carefully stride and pointer parameters:
  - **Stride** → step in bytes between one whole vertex and the next one
    - Example: *sizeof(float)*3+sizeof(float)*2* → size of positions and texture coordinates
  - **Pointer** → offset in bytes of the attribute:
    - Example: *(void*)(sizeof(float)*3)* → offset of texture coordinates after position

Mesh class: indexed geometry

- Indexed geometry avoids **repeat vertices** (important if multiple attributes are used)
- Saving memory is good for performance:
  - Less bytes to travel through **memory bus**
  - With indexes GPUs can use **vertex cache** for recently transformed vertices in vertex shader.



Mesh class: Element Buffer Objects

- For storing indices an Element Buffer Object (**EBO**) must be created.
- Uses same API as VBO but with ***GL_ELEMENT_ARRAY_BUFFER*** as target parameter:
  - glGenBuffers, glBufferData, glBindBuffer and glDeleteBuffers
  - glBufferSubData , glMapBuffer, glMapBufferRange and glUnmapBuffer
- For drawing using EBO, glDrawElements must be called instead of glDrawArrays

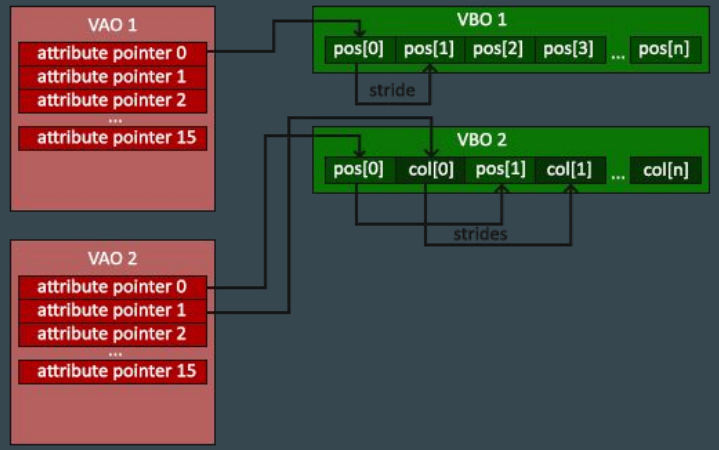
Mesh class: Element Buffer Objects

- Loading EBO example:

```
void Mesh::LoadEBO(const aiMesh* mesh)
{
    glGenBuffers(1, &ebo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);

    unsigned index_size = sizeof(unsigned)*mesh->mNumFaces*3;
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, index_size, nullptr, GL_STATIC_DRAW);
    unsigned* indices = (unsigned*)(glMapBuffer(GL_ELEMENT_ARRAY_BUFFER, GL_MAP_WRITE_BIT);

    for(unsigned i=0; i< mesh->mNumFaces; ++i)
    {
        assert(mesh->mFaces[i].mNumIndices == 3);  // note: assume triangles = 3 indices per face

        *(indices++) = mesh->mFaces[i].mIndices[0];
        *(indices++) = mesh->mFaces[i].mIndices[1];
        *(indices++) = mesh->mFaces[i].mIndices[2];
    }

    glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);
    num_indices = mesh->mNumFaces*3;
}
```

Mesh class: Vertex Array Objects

- Vertex Array Objects (VAO) were created for simplifying and reducing number of function calls for rendering a VBO:
  - glBindBuffer calls for VBO and EBO
  - glEnableVertexAttribArray call for each attribute
  - glVertexAttribPointer call for each attribute.
- VAO stores all of this calls so that can be replaced with one single call ⇒ glBindVertexArray



Mesh class: Vertex Array Objects

- Creating VAO (for separated array attributes) example:

```
void Mesh::CreateVAO()
{
    glGenVertexArrays(1, &vao);

    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,ebo);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, (void*)(sizeof(float)*3*num_vertices));
}
```

Mesh class: Draw

- Mesh drawing example:

```
void Mesh::Draw(const std::vector<unsigned>& model_textures)
{
    unsigned program     = App->render->default_program;
    const float4x4& view = App->camera->GetView();
    const float4x4& proj = App->camera->GetProjection();
    float4x4 model       = float4x4::identity;

    glUseProgram(program);

    glUniformMatrix4fv(glGetUniformLocation(program, "model"), 1, GL_TRUE, (const float*)&model);
    glUniformMatrix4fv(glGetUniformLocation(program, "view"), 1, GL_TRUE, (const float*)&view);
    glUniformMatrix4fv(glGetUniformLocation(program, "proj"), 1, GL_TRUE, (const float*)&proj);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, model_textures[material_index]);
    glUniform1i(glGetUniformLocation(program, "diffuse"), 0);

    glBindVertexArray(vao);
    glDrawElements(GL_TRIANGLES, num_indices, GL_UNSIGNED_INT, nullptr);
}
```

Exercise

- Load **Bakerhouse** model and texture
- Our mesh is static so, create **interleaved** arrays for VBO
- Add a imgui menu showing mesh related info:
  - Num vertices
  - Num triangles
  - Diffuse Texture