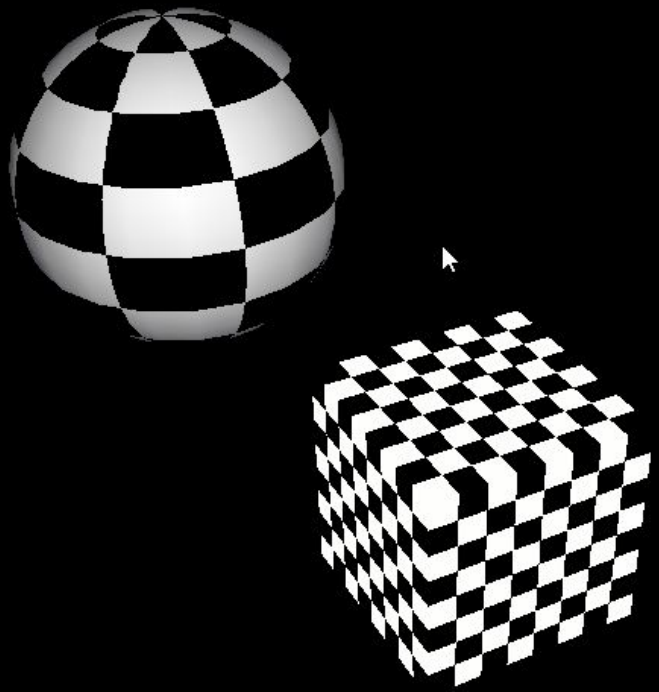
Game Engines

GameObjects & Components

PLAY

Grid Dbg Draw

Options
- RootNode
- camera
- NPC
- NPC 2
- NPC - follower
- Path
- Cube
  - Sphere



Resources

Options
- Textures
- Meshes
- Audios
- Scenes
- Bones
- Animation

Options

Cube

Local Transformation

0.000	2.750	0.000	Position
0 deg	0 deg	0 deg	Rotation
1.000	1.000	1.000	Scale

Bounding Box: 1.00 1.00 1.00

Velocity: 0.00 0.00 0.00 (0.00 m/s)

Geometry

Active Delete Component

Cube Preset

Change Resource

R:255 G:255 B:253 A:255 Tint

12 Triangles (36 indices 16 vertices)

UVs Normals Colors

Potential Bones: 0

Attached to 0 bones

Current: - not assigned -

Pick Another

Material

Active Delete Component

Checkers Texture Preset

Change Resource

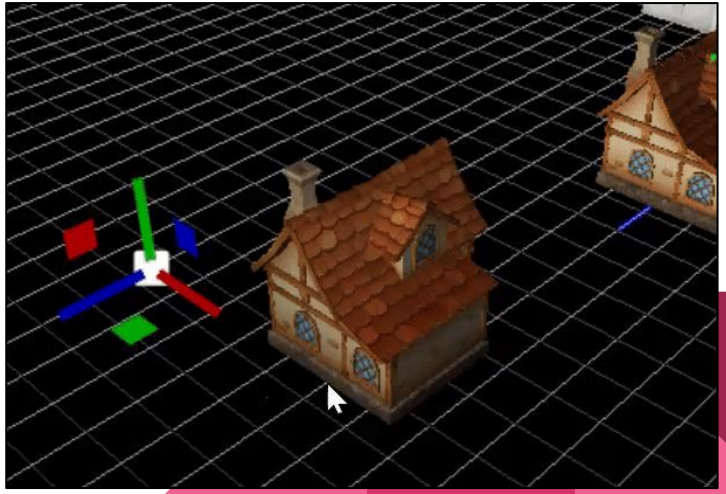
0.000	0.000	Position

Configuration

Options
- Application
- LevelManager
- Window
- Renderer
- Camera
- Textures
- Audio

Component Design Pattern

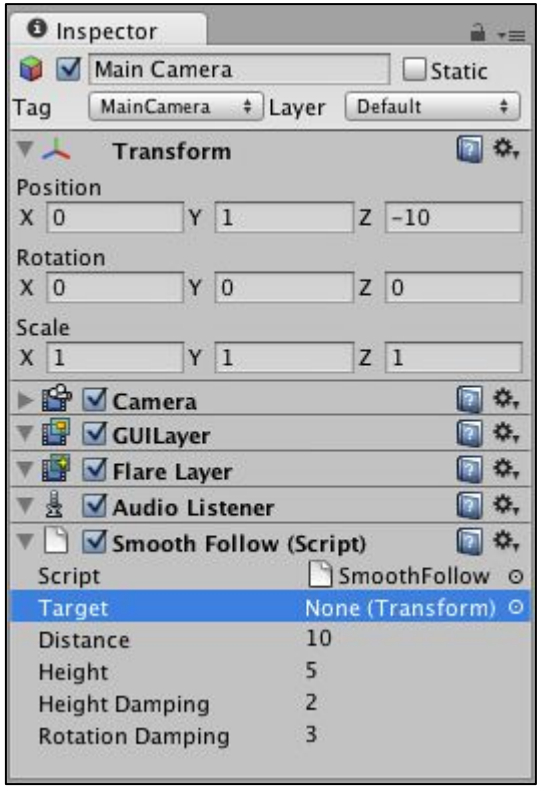
- We will use the Decoupling Design Pattern: Components
- Our goal is to mimic Unity's GameObject structure
- Basically each GameObject contains 0...N components
- Components could be:
  - Transformation
  - Mesh
  - Material
  - Light
- Can I have multiple of each Components ?

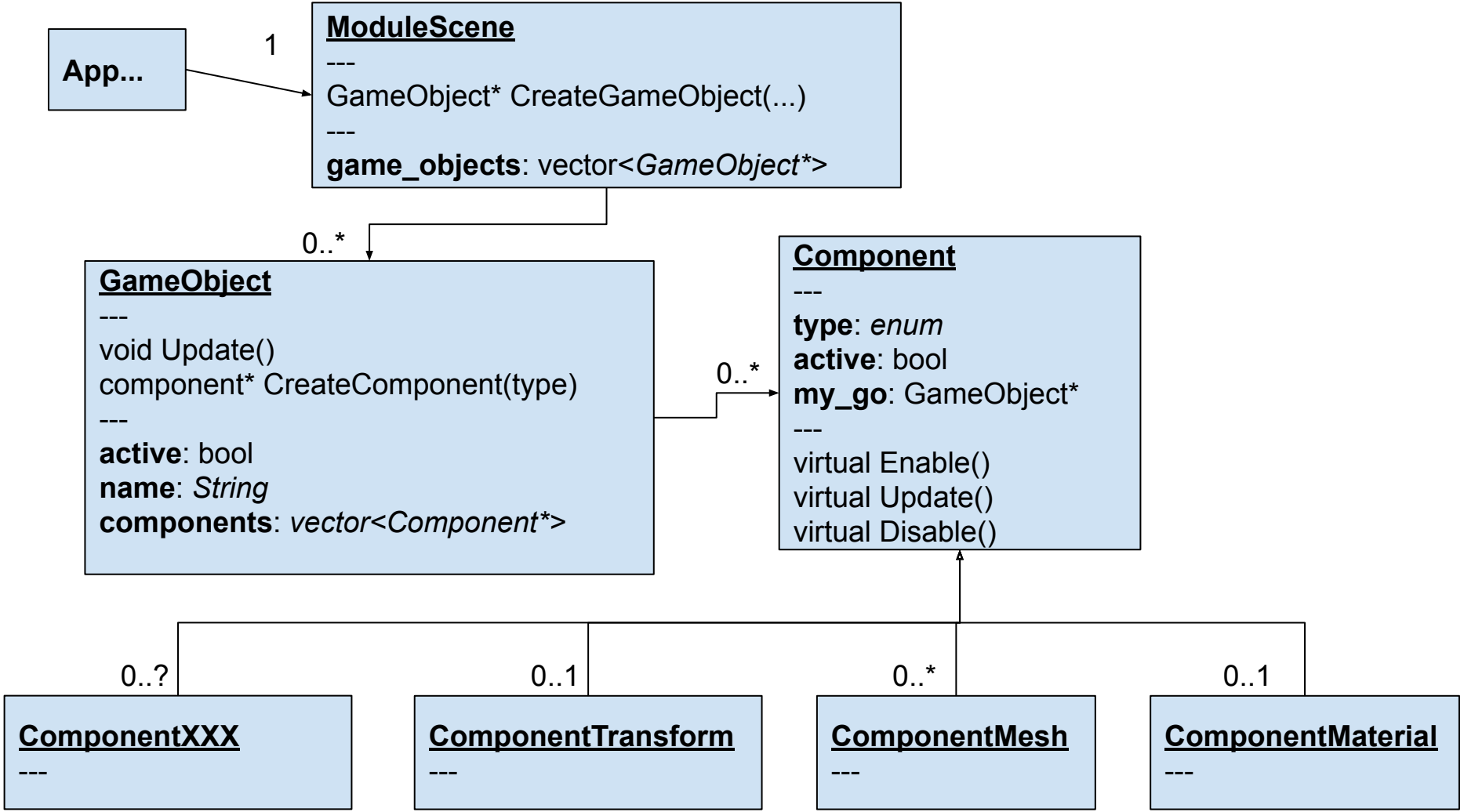


Component Design Pattern

Take a moment to draw the UML for this structure:

- **GameObjects**: properties and methods ?
- **Components**: properties and methods ?
- Relationships ? Type of containers ?
- Where do the different components go ?
- **IGNORE** tree structure for now
- Go!




App...
1
ModuleScene

GameObject* CreateGameObject(...)
---
game_objects: vector<GameObject*>
0..*
GameObject
---
void Update()
component* CreateComponent(type)
---
active: bool
name: String
components: vector<Component*>
0..*
Component
---
type: enum
active: bool
my_go: GameObject*
---
virtual Enable()
virtual Update()
virtual Disable()
0..?
0..1
0..*
0..1
ComponentXXX

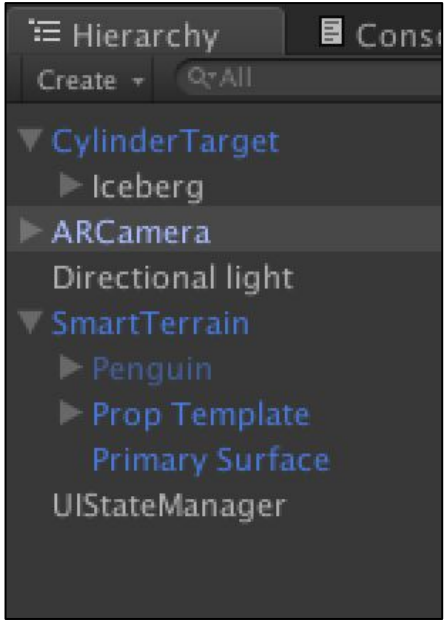
ComponentTransform

ComponentMesh

ComponentMaterial

But … it’s a (Quad) Tree !

- We also need to express the world as a tree!
- Keeps the world organized and transformations pile up
- This will be very relevant when doing animation :)
- Assimp already gives us everything in a nice tree:
  - Start from **aiScene**::*mRootNode* then go recursive from there
  - Then loop **aiNode**::*mNumChildren*
  - … then deal with each **aiNode**::*mChildren*[n]



Calculating the Matrix

Your global matrix = your parent's **global** matrix * your local Matrix

Our approach, from simpler (and slower) to more complex (and better):

1. Recalculate **all** the global matrices at the beginning of the frame
2. When a local matrix is changed, recalculate all the global matrices recursively
3. A better version of this would be lazy evaluation via dirty flags
4. For the optimization freaks… try to move it to iterative ;)

(did anybody say innovation?)

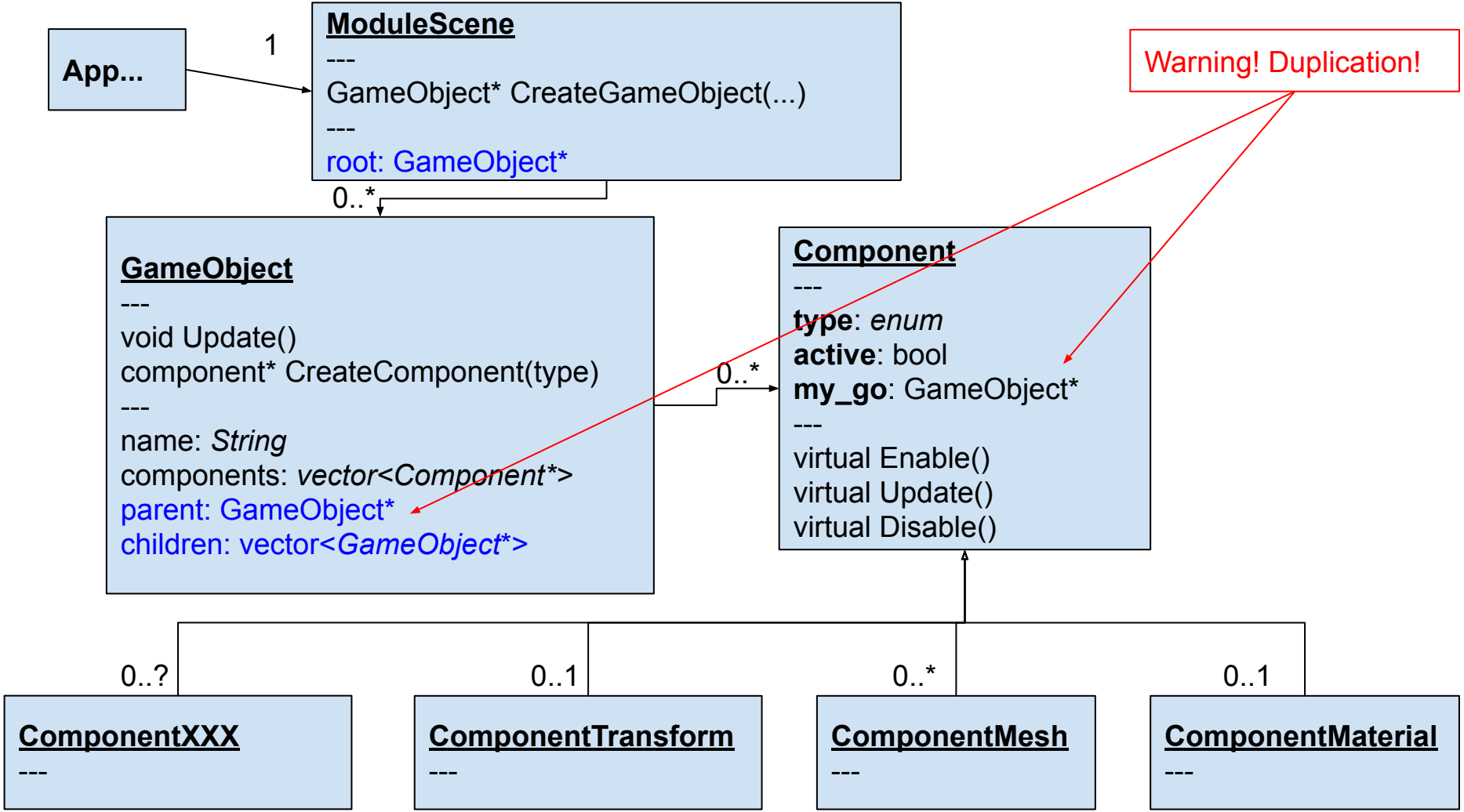
Using the Matrix

Before drawing simply push the Matrix to OpenGL:

```
glPushMatrix();
glMultMatrixf(my_global_transformation_matrix);

… draw everything here

glPopMatrix();
```


App...
1
ModuleScene

GameObject* CreateGameObject(...)
---
root: GameObject*
0..*
Warning! Duplication!
GameObject
---
void Update()
component* CreateComponent(type)
---
name: String
components: vector<Component*>
parent: GameObject*
children: vector<GameObject*>
0..*
Component
---
type: enum
active: bool
my_go: GameObject*
---
virtual Enable()
virtual Update()
virtual Disable()
0..?
0..1
0..*
0..1
ComponentXXX

ComponentTransform

ComponentMesh

ComponentMaterial

Loading transformation

- Break transformation info in position, scale and rotation
- We keep them separated and only mix them to form a Matrix for drawing
- Euler bad, Quaternions good:
  - No gimbal lock
  - Avoid euler standard rotation order madness

```
aiVector3D translation, scaling;
aiQuaternion rotation;

node->mTransformation.Decompose(scaling, rotation, translation);

float3 pos(translation.x, translation.y, translation.z);
float3 scale(scaling.x, scaling.y, scaling.z);
Quat rot(rotation.x, rotation.y, rotation.z, rotation.w);
```

Loading textures from Assimp

- All aiMaterial are stored in aiScene::mMaterials
- Each aiMesh point to it’s material with aiMesh::mMaterialIndex
- Use aiMaterial::GetTexture() to read the filename **(mind the paths!)**

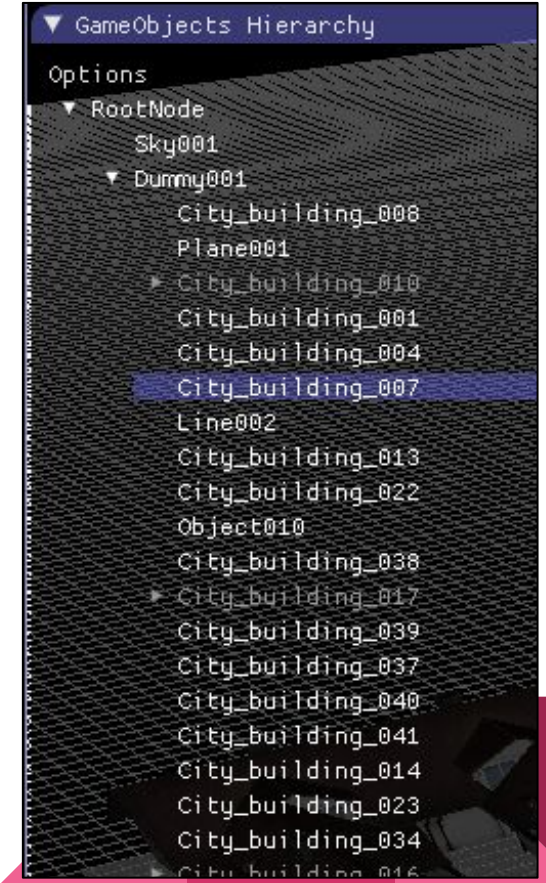
```
const aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
...

aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
uint numTextures = material->GetTextureCount(aiTextureType_DIFFUSE);

aiString path;
material->GetTexture(aiTextureType_DIFFUSE, 0, &path);
```

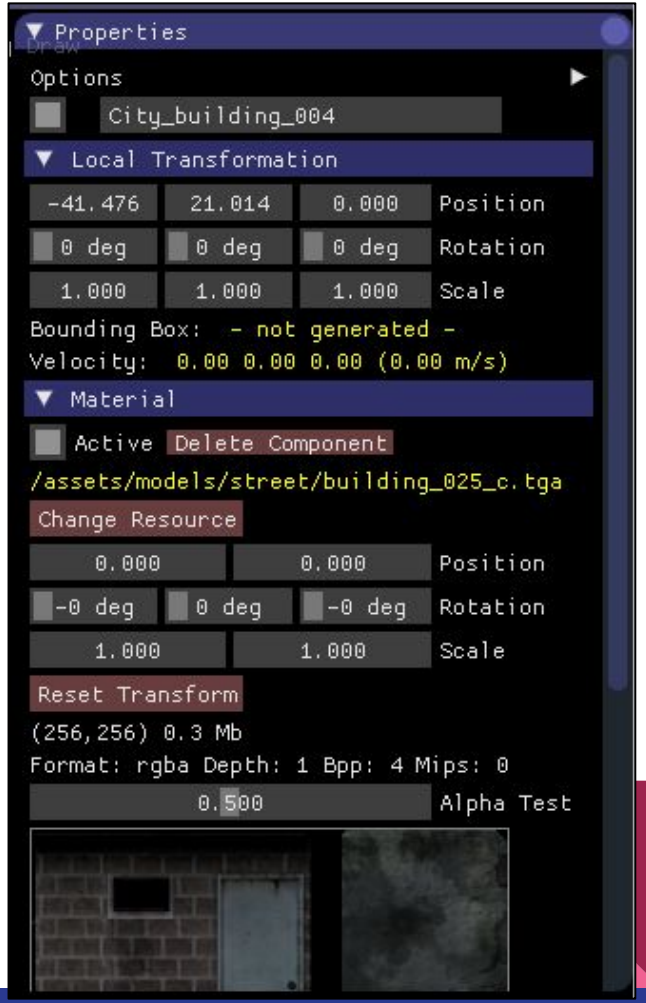
Editor window: The Hierarchy

- Similar to Hierarchy from Unity
  - *Use ImGui::TreeNodeEx() recursively*
- Allow **selecting** and ~~**dragging**~~ to another parent
- Careful with the math! Your local transformation need to be recalculated so the objects stays in the same spot with the same rotation :)
- Allow right click menu on gameobject to:
  - Move Up/Down in the list of childs
  - Create Empty Child Gameobject
  - Destroy



Editor window: The Inspector

- Create also an Inspector window that shows the **selected** GameObject on the Hierarchy window
- You should be able to enable/disable and change the name of the Gameobject
- Using *ImGui::CollapsingHeader* add sections for each component (with a button to enable/disable):
  - Create Transform, Mesh and Material Components
- *Tip*: Components drawing their own panels (?)
  - virtual Component::OnEditor()



References

- Entity Systems, from Object Oriented to Composition
- Break down of Entity Systems (for MMO but equally applicable)
- GameDev article on Composition for Entity Systems
- Article on going beyond and optimizing this structure
- Rotations could be tricky, check this Unity article on how to keep quaternion and euler in sync

Homework Recap

1. Create GameObject class with component Design Pattern
2. Create the Transform, Mesh and Material components
3. Have the ModuleScene contain only the root node
4. GameObjects should have list of children and pointer to parent
5. Load Assimp's hierarchy and info for those components:
    a. Transform
    b. Mesh
    c. Material
6. Create editor support for the hierarchy and properties
    a. Including create new empty gameobjects and components