Assignment 2 – Document Job Task and Scene Graph

Team Road Killer

Weihao Yan, Yun Jiang, Victor Shu

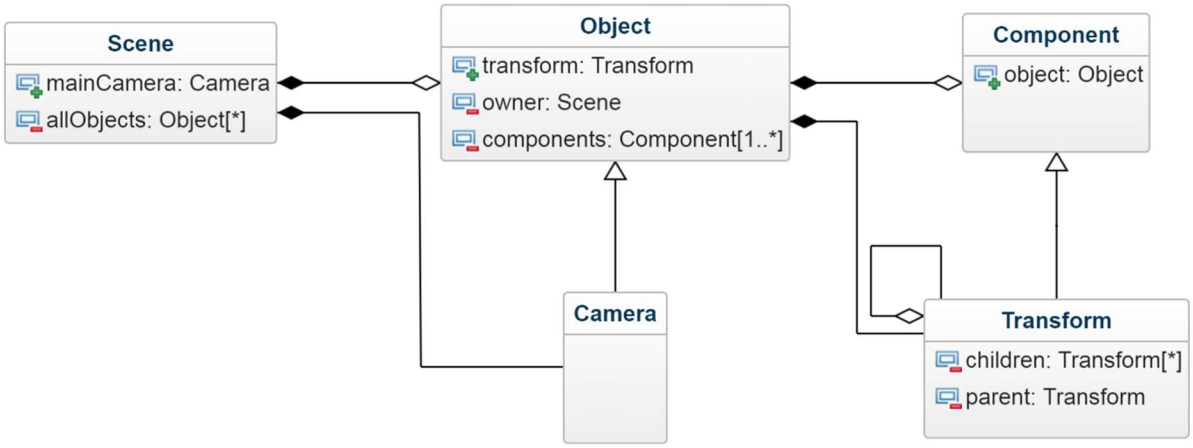
Abstraction

First of all, we don't have a job/task system because we don't have a solid place to use it. Instead of making it just for making it, we decided to focus on our core functionality that is required for a racing game.

In a nutshell, the high-level representation of our scene graph is very Unity-like. Since we don't know how Unity implements their scene graph, we can't say that we are the same. Since we are using an Entity-Component System, we have `Objects` and `Components` attached to them. The `Objects` are stored in an `std::list<Object*>`. One `Object` HAS a special `Component` which is called `Transform` to store parent-child relationship and other information like position, rotation, scale. So, our scene graph is hierarchical, but stored in a planar way.

Data Structure

The classes that are involved in this topic are `Scene`, `Object` and `Component`. The class diagram and the C++ code snippet represent this relationship are listed below.



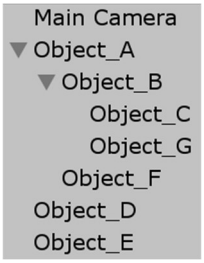
```
class Scene
{
public:
    Camera* mainCamera;
private:
    std::list<Object*> allObjects;
};

class Object
{
public:
    Transform* transform;
private:
    Scene* owner;
    std::list<Component*> components;
};

class Component
{
public:
    Object* object;
};

class Transform : public Component
{
private:
    std::list<Transform*> children;
    Transform* parent;
}

class Camera : public Object
{

};
```

Two things that are not represented in the diagram and the code snippet:

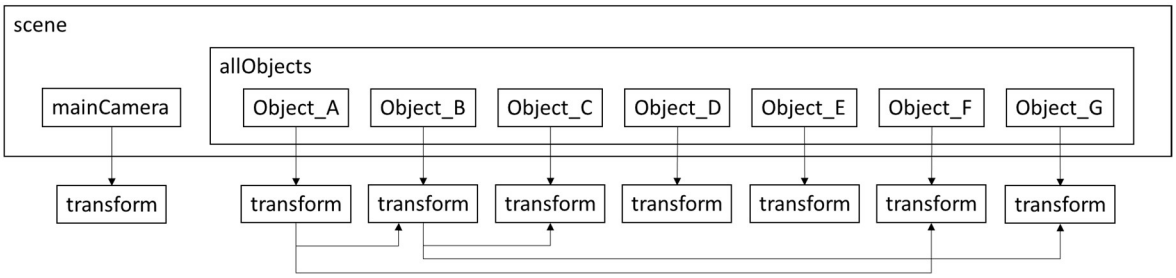
1. `Transform` is a special kind of `Component`, and it is a part of `Object::std::list<Component*> components`. We make it like this so that the user can access the one and the only one `Transform` of an `Object` by calling both `Object::transform` and `Object::GetComponent<Transform>()`.
2. Camera is a special kind of Object, and it is not a part of `Scene::std::list<Object*> allObjects`. Users are supposed to access the main camera by using `App::CurrentActiveScene()->mainCamera`. Other methods like `Scene::GetAllObjects()` or `Scene::FindObjectByName(std::string name)` will not work.

Example:

Consider a scene like this:



The picture shows the hierarchical relationship of the objects, as well as the order of their being added to the scene (the letter that follows the underscore). The actual data stored in memory will be like this:

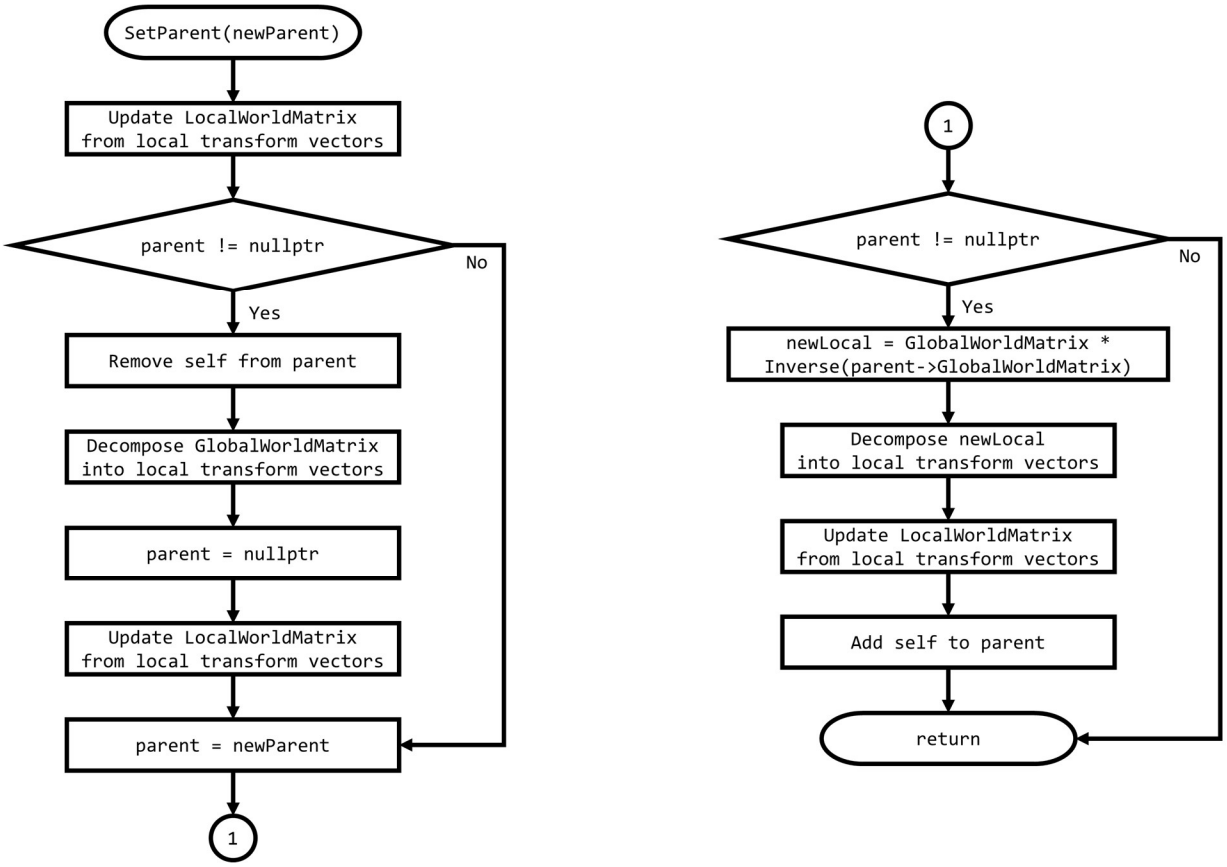


Only child relationship is shown in the picture, but every child holds a pointer to its parent and a transform with no parent holds a `nullptr`. The `mainCamera` can also have a parent or children.

Parenting / Unparenting

If the user wants to set an `Object` as a child of another `Object`, the engine will make sure that the global position, rotation and scale remain the same. The user will need to call `Transform::SetParent(Transform* newParent)` to achieve this. The flow chart of this function is shown below.

The process is made up of two parts. The first part (left side of the chart) will unparent an `Object` which will make it at the highest level of the hierarchy and keep the transform data remains the same globally. The second part will assign the new parent to this `Object` and keep the transform data remains the same.



Update

In the game loop, after physics update and before rendering update, the application will call `Scene::Update()`. Basically, this will do two things: update the camera, and update all objects by calling `Object::Update()`. Since the `Camera` class is derived from `Object`, the scene is just updating all objects. It will update the objects in the order of them being stored, regardless of the hierarchical relationship.

In `Object::Update()`, it will loop through all components it holds and call `Component::Update()`. And since the `Transform` is the only component that manages the parent-child relationship between `Objects`, it is a little special.

When changing the data of a `Transform`, say, `Transform::SetLocalTranslation()`, it will call `Transform::ShouldUpdate()`, which looks like this:

```
void Transform::ShouldUpdate()
{
    for (Transform* child : children)
    {
        child->ShouldUpdate();
    }

    shouldUpdate = true;
}
```

And in `Transform::Update()`, it will check if the `shouldUpdate` member is `true`, and update the matrices according to this.

This ensures that all children of the `Transform` will be updated if itself updates.

*All `Update()` functions mentioned above take `deltaTime` and `totalTime` as input parameters.

System Example: Rendering

In our engine, there are multiple systems. Many of them, if they interact with the `Object`s or `Component`s directly, work in a similar way. We will use our rendering system as an example.

The rendering system is one of the systems of the engine which is responsible for rendering all things in the scene graph. In the engine, we use a special `Component` called `MeshRenderer` to store data that is useful for the system.

The `MeshRenderer` stores two things: `Mesh` and `Material`. One `Object` can have any amount of `MeshRenderer`s, but a `MeshRenderer` can only have one `Mesh` and `Material`, no more no less.

When the rendering system updates – this happens after the scene updates depicted in the former section, it will first gather all `MeshRenderer`s in the scene graph into a list. This part is not optimized since this list is reconstructed every frame even if the data is the same. But since it's not yet the bottleneck we decided to leave it there. After that, the rendering system can use the data of the `MeshRenderer`s in rendering.