Transform Hierarchy Programming Guide

People have been asking me about how to implement the transformation hierarchy, so I'll try to write it here once and for all.

The goal of this part of the project is to be able to place objects relative to other objects. As a demonstration, you should make one object spin around another object.

Hooking up the ParentID (to implement relative transforms)

To implement relative transforms, you can give each Transform in the scene a uint32_t ParentID. The ParentID of a transform is the ID of the transform that this transform is relative to. When a node is a root node (ie. It has no parent), the ParentID is set to -1. This forms a tree of transformations.

When you call AddMeshInstance(), it creates an "Instance" of a mesh, which has a MeshID and a TransformID. For example, if you want to make an instance of a mesh relative to another instance of a mesh, you could do something like:

```
uint32_t newInstanceID1;

AddMeshInstance(scene, meshID, &newInstanceID1);

scene->Transforms[scene->Instances[newInstanceID1].TransformID].ParentID = -1;

uint32_t newInstanceID2;

AddMeshInstance(scene, meshID, &newInstanceID2);

scene->Transforms[scene->Instances[newInstanceID2].TransformID].ParentID =

         scene->Instances[newInstanceID1].TransformID;
```

You can also create a Transform without creating a mesh for it. For example:

```
Transform transform;

transform.Scale = glm::vec3(1.0f);

transform.ParentID = -1;

uint32_t newTransformID = scene->Transforms.insert(transform);
```

Updating the Renderer to handle relative transforms

From there, you need to make your renderer handle this parent hierarchy. In your renderer, you probably already have the following code:

```
glm::mat4 modelWorld;

modelWorld = translate(-curr_transform->RotationOrigin) * modelWorld;

modelWorld = mat4_cast(curr_transform->Rotation) * modelWorld;

modelWorld = translate(curr_transform->RotationOrigin) * modelWorld;

modelWorld = scale(curr_transform->Scale) * modelWorld;

modelWorld = translate(curr_transform->Translation) * modelWorld;
```

This code needs to be extended to handle the parent transforms of the instance. You might do something as follows:

```
glm::mat4 modelWorld;

for (const Transform* curr_transform = transform;

     true;

     curr_transform = &mScene->Transforms[curr_transform->ParentID])

{

    modelWorld = translate(-curr_transform->RotationOrigin) * modelWorld;

    modelWorld = mat4_cast(curr_transform->Rotation) * modelWorld;

    modelWorld = translate(curr_transform->RotationOrigin) * modelWorld;

    modelWorld = scale(curr_transform->Scale) * modelWorld;

    modelWorld = translate(curr_transform->Translation) * modelWorld;


    if (curr_transform->ParentID == -1) {

        break;

    }

}
```

This will accumulate the transforms of each parent of the transform all the way until the root of the tree is reached. After this, your modelWorld transform will represent an absolute transformation of the object in world space, rather than being a relative transform to its parent.

You should be able to test this system by setting the ParentID of a transform to another transform, and you should see that transformations on the parent transform affect its child object as well.

Making one object rotated around another object

Once you have this system working, you should implement one object rotating around another object. I can think of two ways to implement this:

Method 1: Set the child transform's rotation origin to be minus its translation

Method 2: Create an "invisible" transform (ie. not attached to an instance of a mesh) that is a child of the parent, and make your spinning object be a child of that invisible transform. The invisible transform is given a rotation, and the child of the invisible transform is given a translation.

The first method is simple. You might do something like:

```
AddMeshInstance(scene, meshID, &parentInstanceID);

scene->Transforms[scene->Instances[parentInstanceID].TransformID].ParentID = -1;

AddMeshInstance(scene, meshID, &childInstanceID);

scene->Transforms[scene->Instances[childInstanceID].TransformID].ParentID =

       scene->Instances[parentInstanceID].TransformID;

scene->Transforms[childTransformID].Translation = glm::vec3(3.0f, 0.0f, 0.0f);

scene->Transforms[childTransformID].RotationOrigin =

    -scene->Transforms[childTransformID].Translation;
```

The second method might be something like:

```
AddMeshInstance(scene, meshID, &parentInstanceID);

scene->Transforms[scene->Instances[parentInstanceID].TransformID].ParentID = -1;

uint32_t invisTransformID = scene->Transforms.insert(Transform());

scene->Transforms[invisTransformID].Scale = glm ::vec3(1.0f);

scene->Transforms[invisTransformID].ParentID =

         scene->Instances[parentInstanceID].TransformID;

AddMeshInstance(scene, meshID, &childInstanceID);

scene->Transforms[scene->Instances[childInstanceID].TransformID].ParentID =

         invisTransformID;


scene->Transforms[invisTransformID].Rotation =

         glm ::angleAxis(glm ::radians(30.0f), glm ::vec3(0.0f, 1.0f, 0.0f));

scene->Transforms[scene->Instances[childInstanceID].TransformID].Translation =

         glm ::vec3(3.0f, 0.0f, 0.0f);
```

Updating Transforms at Each Frame to Implement Animation

To animate transforms, you should should move them a little bit each frame during the Simulation::Update() function.

For example, you could add a member variable to the Simulation class which is “uint32_t mSpinningTransformID;” During Simulation::Init(), you would set mSpinningTransformID to the ID of a transform which you created during the initialization code, like setting it to the transform ID of the node you want to make spin. Then, during Simulation::Update(), you would use this mSpinningTransformID to refer to the object and update it.

For example, you might do something like… (inside Simulation::Update())

```
float angularVelocity = 30.0f; // rotating at 30 degrees per second

mScene->Transforms[mSpinningTransformID].Rotation *=

        glm::angleAxis(

            glm::radians(angularVelocity * deltaTime),

            glm::vec3(0.0f, 1.0f, 0.0f));
```