TDT4195: Visual Computing Fundamentals

Computer Graphics - Assignment 3

Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)

Bart van Blokland   Michael Gimle   Peder B. Sundt

September 23, 2022

- **Delivery deadline: October 7th, 2022 by 23:59.**
- **This assignment counts towards 6% of your final grade.**
- You can work on your own or in groups of two people. Please note that we have higher expectations in this case.
- Deliver your solution on *Blackboard* before the deadline.
- Use the Gloom-rs project along with all modifications from Graphics Labs 1 and 2 as your starting point.
- Do not include any additional libraries apart from those provided with Gloom-rs.
- Upload your report as a single .PDF file.
- Please use the included shell scripts to automatically pack up the source code correctly into a ZIP file, and upload it alongside the PDF. It should contain everything *except* your `target` and the two files in the `resources` folder we provided as a handout. The final size of the zip will probably be below or near 50kb if you did it correctly, assuming you have not added any new assets not part of the handout.
- All tasks must be completed using Rust and GLSL.
- Use only functions present in OpenGL revision 4.0 Core or higher. If possible, version 4.3 or higher is recommended.
- The delivered code is taken into account with the evaluation. Ensure your code is documented and as readable as possible. We want the final state of the code, you do not have to include the code for intermediate tasks.
- Currently MacOS support for OpenGL is spotty at best. We will not support MacOS in any way, shape or form. We strongly advise you to use Windows or Linux instead, if at all possible.

Questions which should be answered in the report have been marked with a **[report]** tag.

Introduction

This is the final Computer Graphics assignment. As with the second one, start off with your code from the previous assignment.

Individual tasks of the previous two assignments have asked you to implement various pieces of functionality necessary to render a 3D scene. That is, a function to create Vertex Array Objects, the implementation of a controllable camera with an appropriate projection, and the corresponding Vertex and Fragment Shaders.

In this assignment, we're going to put all of that to use, and create a scene that's far closer to something you might find in a modern video game! For starters, we'll be drawing models with far more triangles than the dozen or so you've been drawning thus far. Next, we'll implement some very simple lighting, and finally we'll animate the models using a data structure found in every single major game engine out there, known as a "Scene Graph".

The scene consists of lunar terrain and a helicopter [1]. We'll also – quite literally – make this thing fly.

Some of the things we're asking you to do in this assignment will require you to write some more code than what you're used to from the previous ones. We have therefore done the heavy lifting for you, and have provided a handout containing all the code needed to load the models, and for creating and managing the scene graph. We also packaged up the models themselves.

To extract it into your project:

- Put the contents of the "src" directory in the gloom-rs/src directory. Make sure to run `cargo build` again after you do so!
- Place the "resources" directory in the "gloom-rs" folder that also contains the src and shaders directories.

[1]Yes this is scientifically possible, please stop asking questions.

Introduction to the Scene Graph

As stated before, we will create and animate a larger scene consisting of a number of different objects. One problem with such scenes is that the transformations for individual objects become increasingly difficult to compute. We will focus on the primary way for modelling scenes, and simplifying transformations within them; the Scene Graph data structure.

In its essence, a Scene Graph is a tree-like data structure describing a hierarchy of *nodes* [2]. This might seem like a somewhat vague definition at first *(and you'd be right!)*, but the important bit to note here is the word *hierarchy*.

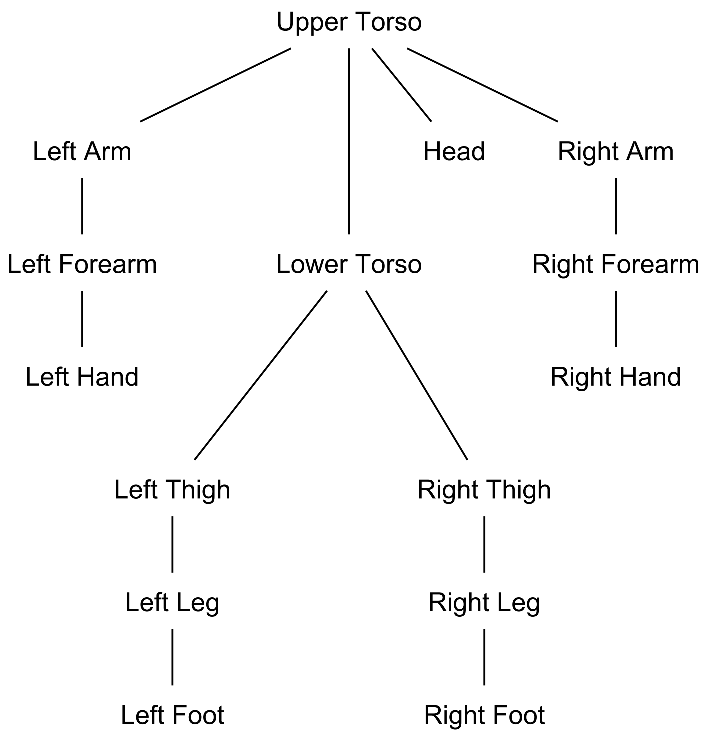
The point behind organising a scene into a hierarchy lies in the fact that it significantly simplifies calculating transformations, and most scenes can intuitively be modelled as such. To illustrate this, let's have a look at an example:

Consider either one of your arms. When you move it around, notice how your forearm moves along with it. Moreover, since the arm and forearm are connected by a joint, it is impossible to move your forearm independently of your arm *(unless something terrible has happened)*.

In this way, the position of the forearm in 3D space can be described relative to the position of the arm. The same is true for the position of the hand relative to the forearm.

It is possible to model the whole human body in a similar way. Considering the upper torso as a reference, a possible hierarchy is shown in the figure below. Note that toes and fingers have been excluded for simplicity.

[2]The best description of the scene graph would be a tree structure: it's a hierarchy of nodes. However, because in some cases it's desirable to reuse child nodes, some nodes can have multiple parents. This means the data structure is technically a directed acyclic graph rather than a tree.



From an implementation perspective it is possible to calculate the entire sequence of transformations of individual parts of a model (such as body parts of a human). However, it is often far easier to calculate transformations of a particular part of a model *relative* to other part(s).

In fact, due to the interdependence of transformations between nodes, a large part of the transformations you'd apply on, for instance a hand, are *identical* to the ones on a forearm. This is precisely what the Scene Graph allows you to exploit.

Additionally, in the case of the movements of your arm, it is far easier to describe how your forearm moves compared to your arm than how it moves relative to the floor. In terms of transformations, the main idea of the Scene Graph is therefore that each node *describes solely how its contents move relative to its parent node*. You can subsequently *combine* the transformations of each parent node (down from the root node of the tree) with the specific child node to obtain the complete transformation of that node.

The real power here is that the child node does not need to care what the parent node(s) are doing. If a character is walking on top of a car, that car can be driving all over the place, but all you need to do for that character is computing its movements relative to the car itself.

There are many ways in which nodes in a Scene Graph can be used. Typically, a Scene Graph Node is implemented only as an “interface”, which can be implemented by different node types with specific specialisations. It is for instance possible to create nodes which focus on setting values of specific uniform variables, or enable and disable shaders used for rendering particular parts of the scene with specific visual effects.

However, our usage of the Scene Graph in this assignment will focus on on how the Scene Graph hierarchy significantly simplifies the process of calculating transformations of objects. As such we’ll only focus on implementing the Scene Graph in its most basic form. This form is equivalent to a *directed acyclic graph*, which include tree structures.

First, we’ll assume there’s only a single type of node. Each node describes its transformation (in a sense movement) relative to its parent in terms of a position, rotation and size (scale). These values will be compiled into a single *model* transformation matrix each frame.

Next, each node has a *Vertex Array Object ID* along with the *number of indice*s, which combined represents the appearance of the node in question. Finally, each node contains a list of child nodes.

One of the source files (scene_graph.rs) that has been attached to this assignment on Blackboard already contain most of the basic functionality needed to implement such a scene graph.

Setting up the Scene Graph

Setting up a Scene Graph is essentially equivalent to constructing a tree data structure, as mentioned previously. You create instances of the data structure of each node, and add each child node to its parent’s list of children.

You also initialise whichever values need initialisation, such as the initial position and VAO ID of the node (as well as others).

Updating and traversing the Scene Graph

Setting up the Scene Graph only needs to be done once. Updating and rendering it is something that can be done each frame. The purpose of updating it is mainly to update positions of animated objects. As with stop-motion animation, the illusion of moving objects can be created by incrementally moving them around a scene many times per second.

It is important here to make a distinction between the two main types of implementing animation: frame-based and time-based. Frame-based animation is by far the easiest to implement. Each frame, you move an object by a specific amount. Assuming the framerate is constant, this gives perfectly acceptable animations.

Unfortunately, this assumption is also the major downside of frame-based animation. First, the framerate at which a scene can be rendered can slow down significantly as the scene becomes more complex, as each individual frame takes longer to draw. This can also depend on hardware. On the other hand, if the hardware is capable of rendering past a specific framerate for which the animation was designed, animations can appear unnaturally fast!

The solution here is to make the speed of the animations depend on a measure which is more consistent: time itself. The idea is to make the displacement of objects depend on how much time has *elapsed* since the previous frame (a function has been provided for this in the handout code). This means that the displacement of objects is greater each frame the more time has elapsed since the previous frame.

The way you commonly implement this is requesting the time that has elapsed since the previous frame. You then scale/multiply the movement distance of your object by the elapsed time.

Traversing the Scene Graph is a matter of iterating over each node in a depth-first-search order, which should be safe since the graph ought be acyclic. The only important bit is that the parent node is evaluated before its children are, because child transformations depend on those of its parent, as described previously. In our case, we'll determine the correct transformation matrix based upon the updated location, scale, and rotation values of the node (see the respective task for detailed instructions). There is no need to store this transformation in the node, as we'll be using it immediately.

Rendering the Scene Graph

Rendering a Scene Graph is a simple extension of the logic needed to compute all the transformations; We render the nodes in the identical order as to how we traverse them. While traversing the nodes we have to make sure to also setup the correct rendering state and issue the draw calls after having computed the proper transformations.

To set the correct OpenGL state we need the complete transformation matrix for each particular node. As the node's *model* transformation matrix depends on the transformations of its parents, and its own transformation matrix (the one which describes the transformations relative to its parent), which is to be multiplied with the one of its parent in the correct order to obtain the accumulated *model* matrix. It's also necessary to apply the view and projection matrices before finally sending them to OpenGL.

This is usually accomplished with a matrix stack. The idea is that each node which requires transformations calculates its relative transformation, multiplies it with the one on the top of the stack, and pushes it on to the stack. Note the word “requires” here, because in practice many nodes in the scene graph do not affect transformations. They can for instance affect render settings or pipeline properties such as lighting coefficients (for phong lighting) or change shaders. Because of that limitation, a stack is essential.

Instead of making this stack by hand, it is often easier to exploit the *program stack* by using recursive function calls. We regard this as the most intuitive way to solve this assignment, as it is often the easiest way to traverse in a graph in depth-first-order.

Note: you’re allowed to make modifications to the supplemental source files, if desired. You may for instance want to add information to each node regarding which shader to use, in case you want to experiment with multiple shaders.

Reference Points

Before we start on the main objective of the assignment, we first should take a detailed look at how movement relative to a parent node can be achieved when using a scene graph.

Specifically, it’s worth noting that objects can be said to move relative to their parent (such as your hand in relation to your forearm), but often the points which objects rotate *around* are not the same. Your hand rotates around a joint at the end of your forearm, while your forearm rotates around a joint in your elbow.

For this reason scene nodes in a scene graph tend to use *reference point*s, which define the origin of a child node relative to its parent.

To illustrate this, let’s focus on a model of a bike, as shown in figure 1. We’ll assume we only want to animate the front wheel, back wheel, the pedals, and the steer (so we can turn the front wheel left and right). We’ll therefore disregard smaller moving components such as the brakes and chain.

Referencing figure 1, notice that two reference points have been marked; a green one for the bike frame, and a yellow one for the back wheel. Let’s assume that all vertices of the bike have been specified relative to the reference point of the bike frame (the green point). Let’s also assume we have stored the vertices of each movable part mentioned previously in separate VAOs.

Notice that the bike’s origin is placed near the ground. Doing so makes it much easier to tilt the bike up or down when climbing up a hill, or riding down one, which is a single rotation about the z-axis.

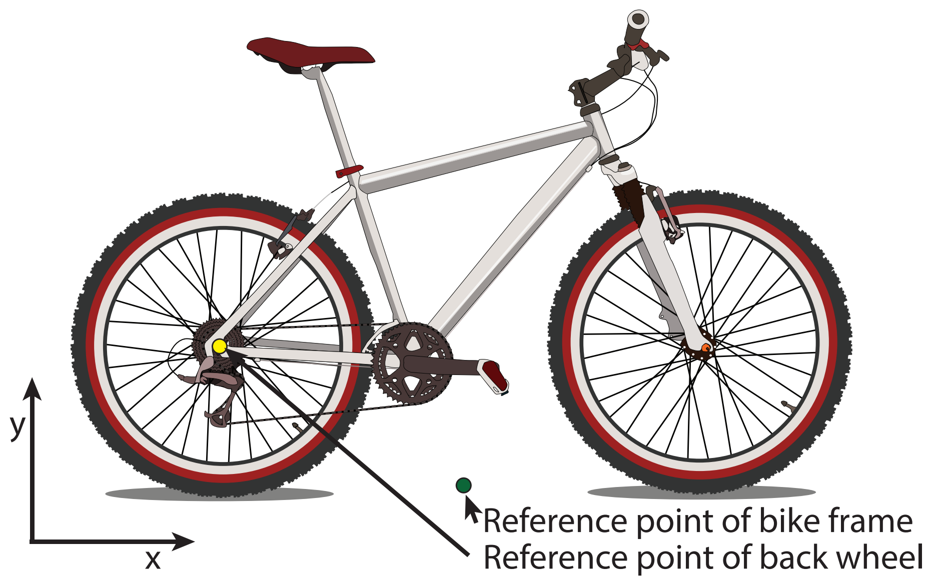


Figure 1: The location of reference points where they could be expected on a 3D model of a bike. Image adapted from: `https://commons.wikimedia.org/wiki/File:Bicycle_diagram-en.svg`

The back wheel's reference point has been placed at the centre of the wheel's rotation axis, so that rotating the wheel can be done by (at some point) performing a rotation around the z-axis.

The main problem here is that vertices in 3D models are commonly referenced relative to the overall origin of the model, rather than the each movable object inside of them. For instance, a vertex part of the bike's back wheel would have a negative x-coordinate and a positive y-coordinate, as the origin it's specified relative to is the green reference point.

This phenomenon is very common because objects are often created or edited in 3D modelling software. These tools will produce models whose coordinates are all specified relative to the *same origin*.

Rotating the back wheel around the reference point of the 3D model yields the effect shown in figure 2. This is generally not how bike wheels behave when rotating.

In order to rotate the back wheel around the reference point, you would first have to move the wheel such that the reference point lies at the origin, then apply the rotation, and finally move it back to where it was. Conveniently, a movement to the origin is accomplished simply by translating it by a vector which is the inverse of the reference point coordinates!

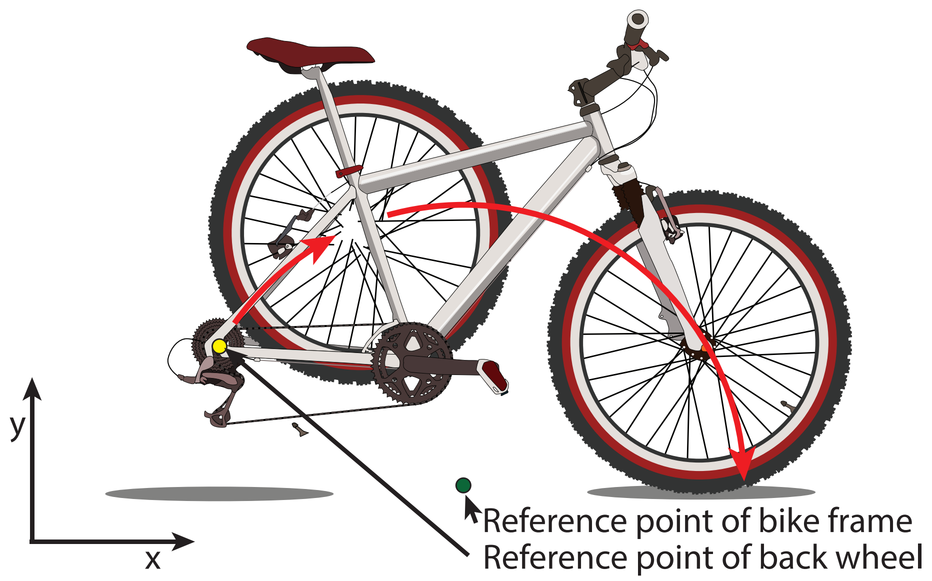


Figure 2: The effect of rotating the back wheel around the origin of the 3D model. The red arrow indicates the rotation motion. Image adapted from: `https://commons.wikimedia.org/wiki/File:Bicycle_diagram-en.svg`

Task 1: More polygons than you can shake a stick at [1 point]

To start out this assignment, we will use the code you wrote in the last assignment to draw a relatively detailed lunar surface model, consisting of close to 100 000 triangles.

You see, modern games don't draw a triangle or two at a time, they draw anything between tens of thousands to millions of triangles. And graphics cards are nowadays powerful enough to be able to do that. So for this first task, you'll get to see what it takes to go from the ~ 5 triangles you've drawn, to a few hundred thousand.

And here's the real kicker: the major difference between drawing a few triangles rather than hundreds of thousands is... *Absolutely Nothing!*

It turns out that what you've already done in the previous assignment is exactly what you need to do in order to draw as many triangles as you want!

Let's dive in, shall we?

a) **[0.4 points]** We'll start out by loading the terrain model.

Since this model covers a fair amount of distance (the difference between the left-most and right-most coordinates is relatively large), you probably want to increase the *far plane* of your perspective matrix before you start. You could for example use `1000.0`.

Up to this point we've been defining individual triangles by manually specifying a vertex and index buffer. In practice, you usually never do this, perhaps apart from specifying shapes of a few triangles at most.

If you look through the handout zip file, you will find a folder named “resources”. This contains two `.obj` files: one for a helicopter (which we'll be using later), and the other for a piece of terrain. Place the folder in the `gloom-rs` folder of your project, next to `shaders` and `src`, so you can access it in a similar way to how you accessed the shaders in the previous assignments.

Load the `lunarsurface.obj` mesh and create a VAO from it, and then draw the VAO. Make sure your call to `gl::DrawElements()` draws the right number of indices!

You won't have to figure out how you can load this model yourself. We've done this for you, and included all the data structures and functions you need in `mesh.rs`:

In order to use this function in `main.rs`, you have to write `mod mesh;` at some point, for example next to the other similar lines in the beginning of the file.

```
fn mesh::Terrain::load(path: &str) -> Mesh;
```

The returned `Mesh` object looks like this internally,[3] which means you'll have access to those fields with the dot operator once you have called the load function shown above:

```
struct Mesh {
    vertices: Vec<f32>,
    normals:  Vec<f32>,
    colors:   Vec<f32>,
    indices:  Vec<u32>,
    index_count:  i32,
}
```

If you want to use another terrain model, you're free to do so, but keep in mind that the model loader that comes with the code handout only supports `.obj` files.

b) **[0.2 points]** The model is going to look pretty boring when colored with a solid color. We want to see some of the juicy details present in the decently high triangle count model. In order to accomplish this, we are going to be adding some *very simple* lighting.

In general, the primary factor determining how much light is reflected from a surface, is the angle between the light source (the sun, a light bulb, etc), and surface itself.

So if we have to compute the color of a given fragment, we first have to know what *angle* it makes with the light source. In order to do this, we have to know what direction the triangle is facing. This is usually achieved by the so/called "normal" vector. This is a unit vector *("unit" meaning "of length 1")* that points towards the "outwards" side of the surface.

Because we know which direction our triangle is facing, we can of course easily compute this. However, the more common way of solving this is to just provide a single normal vector for each vertex in your model. As you may have seen in the Mesh definition in the previous task, the object you load in already contains a list of normals that the `.obj` file defined. Similar to the color buffer in the previous assignment, we now have to make these normals a part of our VAO as well.

Extend your VAO generation function to also take in a vector of floats containing the `x`, `y` and `z` coordinates of the normal vectors included in the mesh. This is basically doing exactly the same as what you did in the second assignment for colors.

[3]Some very minor details are hidden

c) **[0.2 points] [report]** Extend the *vertex* shader to take the new normals as an input, and pass them straight on to the fragment shader through an output variable.

Extend the *fragment* shader to take in the normals that the vertex shader outputs.

After you've done this, visualize the normals for a model by using the `x`, `y` and `z` coordinates of the normals as the `r`, `g` and `b` values for our fragments.

This will make some of the colors negative (i.e. black), but that's fine for this example.

When you do this, your scene should end up looking very colorful. Lots of shades of green, red, and blue. Usually no grays, though.

Position your camera to point into one of the craters in the scene, and attach a screenshot of this immense natural beauty in your report.

d) **[0.2 points] [report]** Finally, we want you to implement some very simple lighting.

In the fragment shader, create a variable holding the direction that the light is coming from. We're creating a light source that's infinitely far away here, which on the moon effectively will be the sun.

```
vec3 lightDirection = normalize(vec3(0.8, -0.5, 0.6));
```

Computing light accurately is immensely complicated and computationally intensive. So instead we're going to use what's known as a "lighting model"; an often crude approximation that looks plausible to the human eye.

The lighting model we are going to be using assumes that every object reflects light equally much in all directions (obviously not the case, but it's easy to compute). This is called the Lambertian model. You can read more about it at `https://en.wikipedia.org/wiki/Lambert%27s_cosine_law`, but it's not required for this assignment.

The equation we want you to use to compute the final color[4] in the fragment shader looks like this, when written with mathematical notation:

$$\text{color}_{\text{RGB}} * \max(0, v_{\text{normal}} \cdot -v_{\text{light direction}})$$

When you set the fragment color to this, all of the details in the lunar surface should become a lot more visible.

Attach a screenshot which shows that the surface is correctly lit.

[4]Note how the equation only applies to the RGB color components! Shadows should not cause the object to become transparent! Try modifying the `gl::ClearColor` to check for this.

Task 2: Helicopter Parenting [1.0 point]

Now that we have a piece of terrain, we can move on to something that moves around in it. Or rather, *flies* around in it! We'll be loading the helicopter model that came with the handout ZIP, and draw it. Also, we'll be constructing the Scene Graph data structure which we talked about before.

a) **[0.2 points]** First of all, load the helicopter model, found in `helicopter.obj`, using the provided function:

```
fn mesh::Helicopter::load(path: &str) -> Helicopter;
```

The returned `Helicopter` object contains the following four items:

```
struct Helicopter {
    body       : Mesh,
    door       : Mesh,
    main_rotor : Mesh,
    tail_rotor : Mesh,
}
```

Create a VAO for every part of the helicopter. Draw the VAOs for the helicopter model. If you haven't done so yet: use `gl::BindVertexArray` to bind a different VAO before drawing it with `gl::DrawElements`. Note that when you bind a VAO with `gl::BindVertexArray`, the previous one is automatically unbound.

b) **[0.5 points]** Now we want to start creating a scene graph. In order to get access to the SceneNode functions, you'll have to write `mod scene_graph;` at the start of `main.rs`. We also recommend adding `use scene_graph::SceneNode;` below it.
In the main function, after you've loaded the models and built the VAOs, construct a scene graph containing:

- A node set up to render your terrain.
- At least one node containing an instance of the helicopter you made in the previous task.

To accomplish this task, do the following:

1) Generate one SceneNode for each object *(one for the Terrain and one for each helicopter part)* in the scene.
2) Organise the objects into a Scene Graph by adding child nodes to their parent's list of children. The organisation must be logical in terms of which object(s) should move relative to other objects, as described previously.

3) Initialise the values in the SceneNode data structure to their respective initial values, such as the position and starting rotations. (Having completed task $3d$ is needed for these positions and rotations to become visible.)
4) Connect the root node of the helicopter to the terrain, and the terrain to a single root node for the entire scene. This is the node you'll be sending into the traverse-and-draw function later.

Relevant functions in the `scene_graph.rs` file:

- `SceneNode::new() -> SceneNode`
  This creates an empty node, for use as the parent of other nodes. Often referred to as root nodes. Empty nodes are not drawable, but its children might be!
- `SceneNode::from_vao(vao_id: u32, index_count: i32) -> SceneNode`
  This creates a node from a VAO ID and the number of indices it should draw from that VAO.
- `helicopter_root_node.add_child(child: &SceneNode)`
  This attaches a node as the child of an another node. Note that this function is called on an actual node, rather than just as a part of the `SceneNode` namespace.
- `helicopter_root_node.print()`
  For the best way to debug^{TM}.

c) **[0.3 points] [report]** Now that we have the scene structured into a *scene graph hierarchy*, we can use it to determine what to draw instead of just calling the draw function for each VAO manually.

We won't be needing the code you wrote to draw the VAOs in the earlier tasks, so you can now get rid of the drawing code in your main loop.

Drawing the scene involves iterating over every `SceneNode` (starting from the root node), binding its VAO and then calling `gl::DrawElements()`.

This snippet should get you most of the way there:

```
unsafe fn draw_scene(node: &scene_graph::SceneNode,
      view_projection_matrix: &glm::Mat4,
      transformation_so_far: &glm::Mat4) {
    // Check if node is drawable, if so: set uniforms and draw

    // Recurse
    for &child in &node.children {
        draw_scene(&*child, view_projection_matrix);
    }
}
```

Note, the `view_projection_matrix` parameter will be used later, but don't worry about it for now. Just pass in the camera matrix you're already computing in the main loop, and call it a day. Also note the `transformation_so_far` parameter. Same story here, simply pass in a $4{\times}4$ identity matrix and move on.

If you use `simple_shader.get_uniform_location("uniform_name")` to locate your uniform variables: try either to (1) pass the shader object in as a parameter to `draw_scene`, or (2) store the shader object as a global variable, (3) define `draw_scene` inside of the rendering loop such that it has access to its closure/scope, or (4) hardcode the location of the uniforms using the `layout(location=X)` qualifier in the vertex shader.

Attach a screenshot showing the helicopter being drawn.

Task 3: The (Model) Matrix: Revolutions [1.5 point]

We've loaded in some models now, and they're looking very pretty with their hundreds of thousands of triangles. But there's a problem here: our helicopter is not moving. How do we go about moving things around the scene?

So far we've seen the *view*[5] and the *projection*[6] matrices, and how they combine to allow us to see the scene from any viewpoint (as we did in the previous assignment). The third matrix commonly used in rendering is called the "model" matrix: it represents the transformations for each of object/model (and parts thereof) *within* the scene. The model matrix is usually applied before the view and projection matrix, hence the name by which the combination of these three matrices are commonly referred to; the Model View Projection (MVP) matrix.

The first thing we need to now is to extend the `draw_scene` function, which already traverses the Scene Graph, to also compute the correct model transformations on-the-fly!

a) **[0.3 points]** In the SceneNode struct (see `scene_graph.rs`), there's a `reference_point` field. This corresponds to the point about which the contents of the node should be rotated. This point should be defined in the coordinate space of the object contained within the node's VAO.

   That basically means you look at the object in the VAO stored in the node, and choose a point which that part should be rotated around.

   Correctly set the reference point of all nodes in your scene graph.
   To avoid a wild goose chase, the tail rotor reference point of the helicopter model is at $[0.35,\ 2.3,\ 10.4]$.

[5]The view matrix transforms objects from world-space coordinates to camera-space coordinates, where the camera is centered in the origin.

[6]The projection matrix transforms from the camera space coordinates to the clip-space coordinates, i.e. the clipbox. This is how we get a sense of depth.

Hint: The origin of the helicopter's model lines up with the main rotor on the xz-plane. If you're unsure about what reference points you may or may not have to change, think about what a reference point does and why we'd need one.

b) **[0.8 point]** Lets have an other look at the `transformation_so_far` parameter in `draw_scene`! This matrix represents the transformation of the parent node: the parent *Model* matrix.

Each time the `draw_scene` traversal function visits a node, make it also compute the current node's relative transformation matrix based on the node's `position` and `rotation` (around the x, y and z axis) fields, and finally combine it with the `transformation_so_far`.[7]

The node should be rotated *(and scaled, which is optional for the assignments)* around its reference point. This requires some other transformations in addition to the rotations themselves. It may be useful to refer back to the introduction of this assignment for an explanation on the theory behind this.

c) **[0.4 point]** The one thing remaining is that when we want to draw a specific node, we need to combine the node's Model matrix, which we just computed, with the scene's *View Projection* matrix.

Therefore, in the `draw_scene` function, combine the View Projection matrix (that you should already be passing around at this point) with the node's model matrix. Pass the final MVP matrix it into the vertex shader as a uniform variable. Again, make sure the multiplication order is correct here!

Hint: If you want to verify that it works, you can change the position or rotation values in the `SceneNode` of the helicopter body.

[7]You may refer to the bottom of the *tiny rust cookbook* from ex1 for an overview of the GLM matrix functions, and how to compose them in a readable fashion.
`https://pbsds.github.io/tiny-rust-cookbook/cookbook_rust.pdf`

Task 4: Spinning into gear [1 point]

Time for some animations! Now that you have set up the scene graph properly, this should be a cinch. All animations should be done in a time-based rather than frame-based way. This is easy if you use the value we already compute at the start of each frame, `elapsed`, which tracks the time in seconds since the first frame! It looks like this:

```
let elapsed = now.duration_since(first_frame_time).as_secs_f32();
```

a) **[0.4 points]** Make the helicopter's main and tail rotor spin in their sockets continuously. The `elapsed` and `delta_time` variables in the main loop are helpful here.

   The order in which you multiplied the matrices in the previous task when drawing the scene graph is very important. If you're not sure whether you have done it right or not, try working through the next subtask, as it should look very obviously wrong if your order is incorrect.

b) **[0.6 points]** For this task, you should have your helicopter follow some animated path. We have provided a function which will generate some coordinates and rotations following a circuit.

   You're free to come up with any animation you'd like here, but it's important that you showcase rotation along all three axes if you do. This animation is going to look wrong unless you have correctly implemented all parts of the Scene Graph update.

   The function lies in `toolbox.rs`, and you can import it much like how you did earlier with `mesh.rs` and `scene_graph.rs`.

   The function is called `simple_heading_animation`, and does some maths to produce a `Heading`, which is a collection of positional and rotational energy.

   ```
   toolbox::simple_heading_animation(time: f32) -> Heading
   ```

   A `Heading` contains the following:

   ```
   struct Heading {
       x     : f32,
       z     : f32,
       roll  : f32,
       pitch : f32,
       yaw   : f32,
   }
   ```

   Hint: The XYZ axes of the helicopter model is *not* aligned to those commonly used in aeronautics and mechanical design.

   Hint: The animation is going to look a little *off*, due to us so far using extrinsic euler angles instead of intrinsic angles. To partially mitigate this we suggest first applying Z rotation, then the Y rotation, then the X rotation.

Task 5: Help! My lighting is wrong! [1 point]

a) **[0.2 points] [report]** Notice anything weird about how the helicopter is lit as it moves around? The lighting doesn't change even though the helicopter is turning!

This is because the normal vectors are defined in relation the model as if the model still had its original orientation. If we want the shadow to move across the helicopter when the helicopter rotates, we have to rotate the normals in the same way that we rotate the model.

Attach two screenshots staken from the same camera position, one where we see the brightly lit side of the helicopter, and another taken when it has rotated and we see a darker side instead.

b) **[0.4 points]** In the real world, solving this problem requires the use of something called a Normal Matrix, but since that's outside of the scope of this course, we'll use a very cheap imitation instead. In short, the idea is that we butcher our existing model matrix to compute the transformation matrix that rotates the normals to the correct orientation.

Note that we need to rotate normals using the Model matrix only; we don't want a matrix that also has the View and Projection matrices applied on it. Fortunately, we have been passing around the View Projection matrices separate from the Model matrices in our traversal function that draws the scene graph!

As such, we can compute two separate matrices, and pass each of them each in as their own uniform variable into the vertex shader:

- The Model View Projection matrix, used to transform the vertex positions
- The Model Matrix, used to transform the vertex normals

c) **[0.4 points] [report]** Before we multiply the Model matrix with the vertex normal, we'll need to make some minor modifications to it to ensure it does not scale or translate the normal vector.

We do this by taking the top 3x3 part of the Model matrix, which gets us all of the rotation and scale applied to the model, and none of the translation. We then multiply this by the normal vector, which results in it being rotated and scaled by the same amount that the object it's attached to would be. Finally, we normalize the result, since the normal vectors should always be of unit length *(i.e. have a magnitude equal to 1)*. This has the effect of reverting any uniform scaling present in the Model matrix.[8]

Attach two new screenshots taken from the same camera position, where we see two sides of the helicopter.

Hint: `normalize` will convert a vector to unit length.

[8] If you scale X, Y and Z independently, we may end up having a problem however...

Hint: `mat3` will convert a 4×4 matrix into 3×3, simply by cropping the upper region.

Hint: Don't forget that you can always visualize the normals using RGB like we did in a previous task to get a more clear picture of what's going on!

Task 6: Time to turn this thing up to ~~11~~ 5 [0.5 point]

a) **[0.5 points] [report]** Everything you have done so far should be easily extendable to be done in loops, so that's what we're going to do for the final part of this assignment.

Instead of creating a single helicopter, instantiate at least **5** helicopters that share the same VAOs / meshes. Keep track of the relevant scene nodes and animate them in the main loop. Every helicopter should have a rotating tail rotor and main rotor, as well as follow some kind of path like in the previous task.

Every one of the helicopters should follow the same path, but none of them should collide with any other. You can accomplish this by feeding the animation function a carefully selected offset.

Attach a screenshot of the five helicopters.

Hint: You can use a `Vec<scene_graph::Node>` to store the helicopter root nodes. You may also store the "main" and "tail" rotor nodes in separate `vec`tors, or reach them via the `.get_child` member function or by simply using square brackets on the `SceneNode`.

Hint: can a scene node be used multiple times?

Task 7: Optional Challenges [At most 0.51 points]

This task is optional. These questions are meant as further challenges or to highlight things you may find interesting. They can reward up to 0.51 points, to supplement missing points from other parts of the assignments. Please show us in the **[report]** how you implemented it, or some eye candy, if you manage to answer some or all of them!

a) Implement Phong shading instead of Lambertian shading **[0.2 points]**

The Phong lighting model is a more visually interesting and more realistic alternative to what we have used in this assignment. It's comprised of three parts, where the lambertian diffuse shading model we used is one of them. If you want a good challenge with coordinate spaces, shaders, and vectors, I recommend giving this a try.

b) Make one of the helicopters controllable **[0.1 points]**

It should be controllable in the same way the camera was in assignment 2 if one did the bonus task, so going forward should move the helicopter in the direction it's facing. Having the helicopter tilt somewhat realistically is recommended, but not required. If doing this task, we suggest you either leave the camera staying still, or implement the chase camera below (other advanced camera control functionality can also be rewarded).

c) Implement a chase camera **[0.2 points]**

A chase camera has a position, a target and a chase radius.

It will constantly look at its target, and if the target leaves the chase radius, it will move towards the target in such a way that the target (barely) reenters the chase radius. Changing the animation function for this task is not required, but could be a good idea in order to showcase the efficacy of your chase camera.

d) Make the door animated **[0.1 points]**

The door of the helicopter is a separate object. This means that it should be pretty easy to animate. Make one of the keys on the keyboard open the door.

You can simply slide it in the Z-direction, but we're very open to see some creative solutions here!

Closing the door again is optional.

e) Use intrinsic rotations **[0.05 points]**

So far we've most likely been using *extrinsic* rotations when transforming both the camera and the models in the scene. Extrinsic rotations are rotations about the XYZ axes of the original coordinate system, which remains fixed / motionless.

This may result in weird behaviour. Try applying the pitch of the camera before you apply the yaw of the camera, then angle the camera 90° to the left and try looking down. You will observe that the world instead seem to roll instead of pitching.

Objective: Rotate the camera and the the objects in the scene graph using *intrinsic* Euler angles, instead of extrinsic ones. Intrinsic rotations rotate about the axes of rotations along with the rotating coordinate system, i.e. along with the moving body. If you implement intrinsic rotations correctly, then it won't matter which order you apply the pitch, roll and yaw.

f) Find the easter egg **[0.01 points]**

Attach a screenshot of the easter egg we have hidden somewhere in the scene.

g) Impress us! **[0.0-0.5 points]**

This is a wildcard for us to give you bonus points if you do anything that's really cool or beautiful. These points are not cheap, but not unachievable either!