TDT4195: Visual Computing Fundamentals

Computer Graphics - Assignment 3

October 31, 2016

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

- **Delivery deadline: November 8th, 2016 by 22:00.**
- **This assignment counts towards 3% of your final grade.**
- You can work on your own or in groups of two people.
- Deliver your solution on *itslearning* before the deadline.
- Use the Gloom 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.
- Upload your report as a single PDF file.
- Upload your code as a single ZIP file solely containing the *src* and *shaders* directories found in the Gloom project. All modifications must be done to files in these directories.
- Your code must be runnable on the lab computers at IT-S 015.
- All tasks must be completed using C++.
- 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.

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 lab, we will use these to create an animated scene of a solar system, using a data structure known as the "Scene Graph".

Introduction to the Scene Graph

In this final lab we will look at how transformations are combined to create and animate larger scenes consisting of a number of different objects. For this purpose we will focus on a specific data structure most commonly used to model scenes; the Scene Graph.

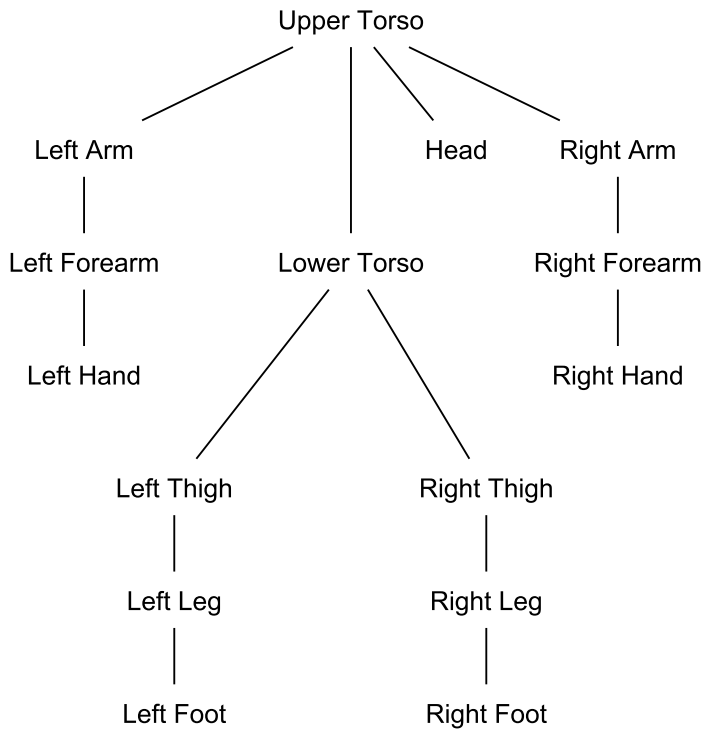
In its essence, a Scene Graph is a tree-like data structure describing a hierarchy of nodes. This might seem like a somewhat vague definition at first (and you'd be right), but the important bit to note here is the word *hierarchy*.

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 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.



From an implementation perspective it is possible to calculate 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). This is the notion which the Scene Graph attempts to exploit.

For instance, 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 how its contents move *relative to its parent node*.

There are many ways in which nodes in a Scene Graph can be used. It is for instance possible to create nodes which specialise in setting values of specific uniform variables, or enable and disable shaders used for rendering particular parts of the scene.

However, in this assignment we'll focus 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 tree structure.

First, we'll assume there's only a single type of node. Each node describes its movement

relative to its parent in terms of a position, rotation and size (scale). As we'll be animating this scene, we'll also keep track of the speed at which the object is moving. These values will be compiled into a current transformation matrix each frame, also stored in the node.

Next, each node has a Vertex Array Object ID, which represents the appearance of the node. Finally, each node contains a list of child nodes.

Two of the source files (SceneGraph.hpp and SceneGraph.cpp) which have been attached to this assignment on It's Learning already contain most of the basic functionality needed to implement a scene graph. Documentation about the functions and data structures it contains are available at the bottom of this assignment text.

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 of the node.

Updating the Scene Graph

Setting up the Scene Graph only needs to be done once. Updating and rendering it is something 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 frame-based and time-based animation. 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 animations depend on a measure which is more constant; 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). This means that the displacement of objects is greater each frame, the more time has elapsed since the previous frame.

Updating the Scene Graph is a matter of iterating over each node in a depth-first-search order, except the parent node is evaluated before its children.

In our case, also determine the correct updated transformation matrix based upon the updated location, scale, and rotation values.

Rendering the Scene Graph

Rendering a Scene Graph is a process almost identical to updating it. Nodes are traversed in an identical order, but instead the correct rendering state is set up each time, and a draw call is issued.

Part of this OpenGL state is the complete transformation matrix for the particular node. As the node's transformation matrix depends on the transformations of its parents, its own transformation matrix (the one which describes the transformations relative to its parent), must 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 sending them to OpenGL.

This is usually accomplished with a matrix stack. The idea is that each node which requires transformations (normally not all nodes do so, although in our case they do), a node takes its relative transformation, multiplies it with the one on the top of the stack, and pushes it on to the stack.

Functions have been provided for a stack-based implementation, though passing arguments through recursive function calls may be possible in the case of this assignment too.

Note: it is allowed to make modifications to the supplemental source files, if desired.

Task 1: Repetition [0.5 point]

a) **[0.2 points]** Two of the four source files attached to this assignment on It's Learning (Sphere.hpp and Sphere.cpp) contain a basic implementation of a function which generates a three-dimensional sphere.

i) Extend the function in Sphere.hpp to also generate a colour buffer (such that each vertex is given its own colour).

ii) Integrate this function to work with your own code. Specifically, the function you implemented in assignment 1 and extended in assignment 2, which converts a vertex, colour and index array to a VAO.

Note that the sphere generating function works in terms of layers and slices. It generates vertices at intersections between each slice and layer, and combines those into triangles. An example of these is shown here:

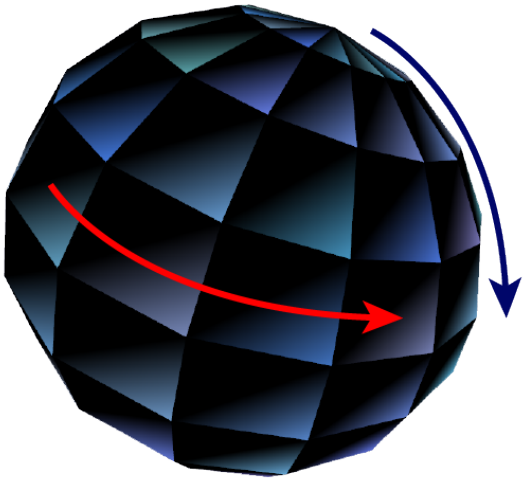


Figure 1: A sphere drawn using slices and layers. The red arrow indicates slices, the blue arrow layers.

Optional challenge: Generate an additional Vertex Attribute for the sphere to include surface normals. Use these normals to add Phong shading to your shader pair for a really cool effect when the light source is located in the centre of the sun.

b) **[0.3 points]** Explain each of the following terms, and their purpose in the OpenGL rendering pipeline:

i) The z-buffer

ii) Framebuffer

iii) Viewport

Task 2: Setting up the Scene Graph [0.5 point]

Create a function which constructs and returns a scene graph containing a solar system. The system must at least contain a sun, around which must revolve at least 5 planets, of which at least two must have at least one moon.

Some notes and requirements about this solar system:

- All bodies (sun, planets, moons, etc) in the solar system may be modelled as spheres.
- You may assume all orbits to be perfectly circular.
- All planets must have different colours.
- Planets and moons must orbit at different speeds, some clockwise and others counter-clockwise.
- The radii of planet orbits must be different from each other.
- Planets and moons must be smaller than the sun. Moons must be smaller than planets.

To accomplish this task, do the following:

i) Generate one SceneNode for each object (Sun, Moon, Planet) in the scene.

ii) 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.

iii) Initialise the values in the SceneNode data structure whose values do not change (such as the rotation speed and direction).

iv) Return a reference to the root node (the SceneNode representing the Sun).

Relevant functions in the SceneGraph.hpp file:

- `float random()`
- `SceneNode* createSceneNode()`
- `void addChild(SceneNode* parent, SceneNode* child)`
- `void printNode(SceneNode* node)`

Task 3: Traversing and updating the Scene Graph [1 point]

Implement a function which updates the Scene Graph.

The aim here is to visit each node in the scene graph, and update the node's position and rotation (whichever are applicable). The traversal order has been described previously. All Planets must rotate around the sun at constant speed. That is, the speed at which they move must not depend on the framerate. You should use time-based animation to accomplish this, which has been described previously. The same counts for Moons.

Next, generate a transformation matrix which represents the updated transformation of the node relative to its parent. Store this matrix in the `currentTransformationMatrix` field in the SceneNode.

Relevant functions in the SceneGraph.hpp file:

- `float getTimeDeltaSeconds()`
- `void printNode(SceneNode* node)`

Hint: The `getTimeDeltaSeconds()` function returns the elapsed time in seconds since the previous time the function was called. Make sure to call it exactly once per frame to obtain the elapsed time since the previous frame.

Hint: (average) speed is defined as travelled distance over elapsed time. If you know the elapsed time, you can calculate the travelled distance. This also works for angles.

Task 4: Rendering the Scene Graph [1 point]

a) **[0.6 point]** Traverse the Scene Graph, similar to the way you did in Task 3. For each visited node:

   i) Set up any applicable OpenGL state necessary to render the celestial body contained in the SceneNode.

   ii) Determine the Model Matrix of the Scene Node. This is done by combining the node's relative transformation matrix (updated during the previous task) with the accumulated matrices of its parents. Lecture 6 discussed how to combine these matrices in more detail. The transformation of any children of the node should utilise the resulting accumulated Model Matrix.

   iii) Calculate the node's MVP matrix. The View matrix should be the one you generated for implementing the camera behaviour in assignment 2. The Projection matrix should also be the one you used in assignment 2. The Model Matrix is the accumulated model matrix mentioned above.

   iv) Render the node's VAO.

b) **[0.2 point]**[**report**] Put two screenshots in your report showing the scene you created from two different angles at different points in time.

c) **[0.2 point]** [**report**] You should at this point have a working solar system which is capable of approximating the orbital motions of a real solar system by modelling it as a Scene Graph.

   What we haven't look at yet in this assignment is how to apply a Scene Graph on to more complex 3D models. In our situation, the sun represented the centre point for the orbits of the planets around it. Likewise, the planets themselves represented the centre point for the orbits of the moons.

   However, in a practical context this is not always the case. Objects may move, rotate or scale around a different point than what could be considered the "location" of the parent object.

   To illustrate this, let's focus on a model of a bike, as shown in figure 2. 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.

   i) Give a Scene Graph structure (similar to the one of the human body found in the introduction of this assignment) of the listed movable components, as well as the bike frame itself showing how you would organise these into a Scene Graph. Briefly motivate your answer.

   Referencing figure 2, 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 triangle

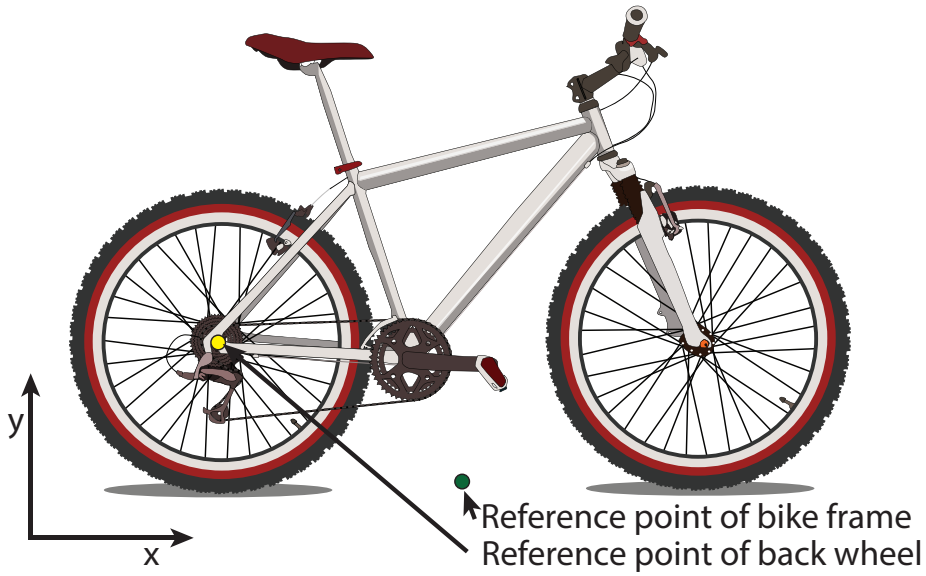


Figure 2: The location of reference points where they could be expected on a 3D model of a bike. Image adapted from: `https://commons.wikimedia.org/wiki/File:Bicycle_diagram-en.svg`

coordinates 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 VAO's.

Notice that the bike frame's reference point 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 along the z-axis.

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 problem comes in when the vertices within 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 located at the reference point of the back wheel would have a negative x-coordinate and a positive y-coordinate when the "origin" or "zero" of the object is located at the bike's (green) reference point. Remember how the origin of the 3D spheres generated previously is at the centre of that sphere.

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

ii) Give a sequence of affine transformations which rotates the aforementioned back wheel of a bike model around the given (yellow) reference point by 30 degrees

around the z-axis. The sequence of transformations should be those a Scene Node of the back wheel would apply to animate the wheel.

It is only needed to do this correctly in two dimensions. For each transformation, only give the type, axis and amount or direction, whichever apply. Giving 4x4 transformation matrices is not necessary.

Assume that all vertices of the bike model are specified relative to the green reference point. The location of the yellow reference point relative to the green reference point is (-5, 3). The location of the bike's Scene Node in the scene is (100, 27).

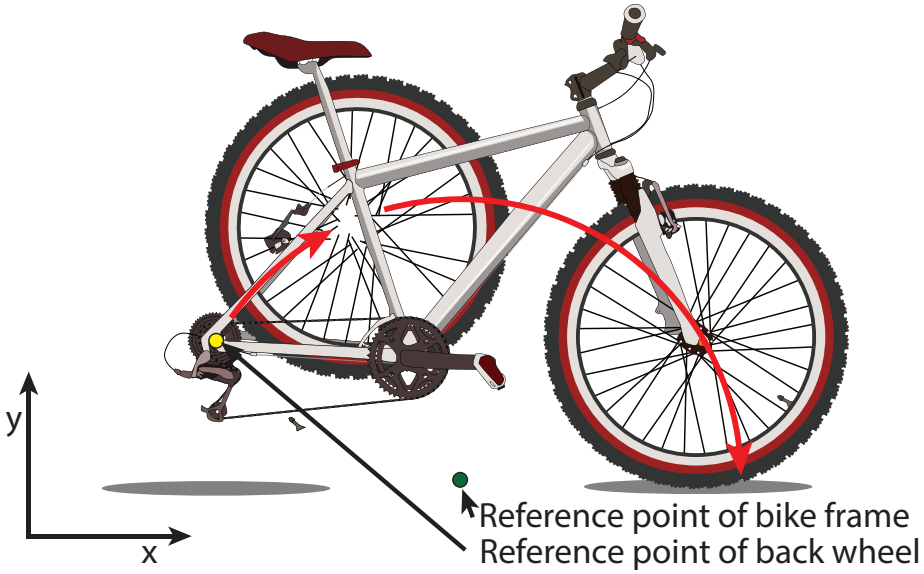


Figure 3: The effect of rotating the back wheel around the origin of the 3D model. The red arrow indicates the rotation motion. Image adapted from: `https://commons.wikimedia.org/wiki/File:Bicycle_diagram-en.svg`

Task 5: Optional Feedback [0 points]

a) **[0 points]** This question is optional and has no effect on your final grade in any way.

For this year's Computer Graphics part of the course, we have created both new assignments and slides from scratch. If we are to improve these for future iterations of this class, we rely on your feedback to improve them.

If you have anything to comment about them, such as what was good, or what was bad, we'd love to hear from you.

Simply leaving an overall score for each would already help a lot. Thanks in advance!

Appendix: Functions available in the SceneGraph.hpp file

```
std::stack<glm::mat4>* createEmptyMatrixStack();
```

Allocates a new, empty matrix stack.

```
void pushMatrix(std::stack<glm::mat4>* stack, glm::mat4 matrix);
```

Pushes a matrix on to the top of the matrix stack.

```
glm::mat4 popMatrix(std::stack<glm::mat4>* stack);
```

Removes the matrix on top of the matrix stack, and returns the removed value.

```
glm::mat4 peekMatrix(std::stack<glm::mat4>* stack);
```

Returns the matrix currently at the top of the matrix stack.

```
void printMatrix(std::stack<glm::mat4>* stack);
```

A debug function which prints the contents of the matrix stack to stdout.

```
struct SceneNode {
        std::vector<SceneNode*> children;

        float rotationX, rotationY, rotationZ;
        float x, y, z;
        float scaleFactor;

        float rotationSpeedRadians;
        glm::vec3 rotationDirection;

        glm::mat4 currentTransformationMatrix;

        int vertexArrayObjectID;
}
```

A data structure representing a single node in the Scene Graph. It contains (in the respective order in which the fields are listed above):

- A list of all child nodes (if any).
- The rotation of the contents of the node around each axis.
- The location of the contents of the node along each axis.
- The size (scale) of the contents of the node along all axis.
- The speed at which the contents of the node rotate along a central point, measured in radians
- A normalised vector determining the direction around which the contents of the node rotate.
- The calculated transformation matrix based upon the location, rotation and scale definitions of this node.
- The OpenGL ID of the Vertex Array Object containing the draw buffers for this object. If this node does not contain an object that should be rendered, this value is set to -1.

```
void addChild(SceneNode* parent, SceneNode* child);
```

Appends a child node to the end of the parent's child list.

```
void removeChild(SceneNode* parent, SceneNode* child);
```

Removes a child from a parent node, if it is present.

```
void printNode(SceneNode* node);
```

A debug function which prints out the values of the fields of the given SceneNode.

```
SceneNode* createSceneNode();
```

Creates an empty scene node. In the order in which the individual fields of the SceneNode data structure are listed above, the fields are initialised as follows:

- And empty list
- The rotations along each axis are set to 0
- The locations along each axis are set to 0
- The scale is set to 1.
- The rotation speed is set to 0.
- The rotation axis is set to the y-axis
- The calculated transformation matrix is set to the identity matrix.
- The Vertex Array Object ID is set to -1.

```
float random();
```

Returns a random floating point value between 0 and 1. Can be useful for creating variations in planet or moon sizes, planet colours, orbit radii, and so on.

```
double getTimeDeltaSeconds();
```

Returns the time in seconds since the *previous time this function was called*. You should call this function once per frame.