Assignment 1 — Solar System

Lund University Graphics Group

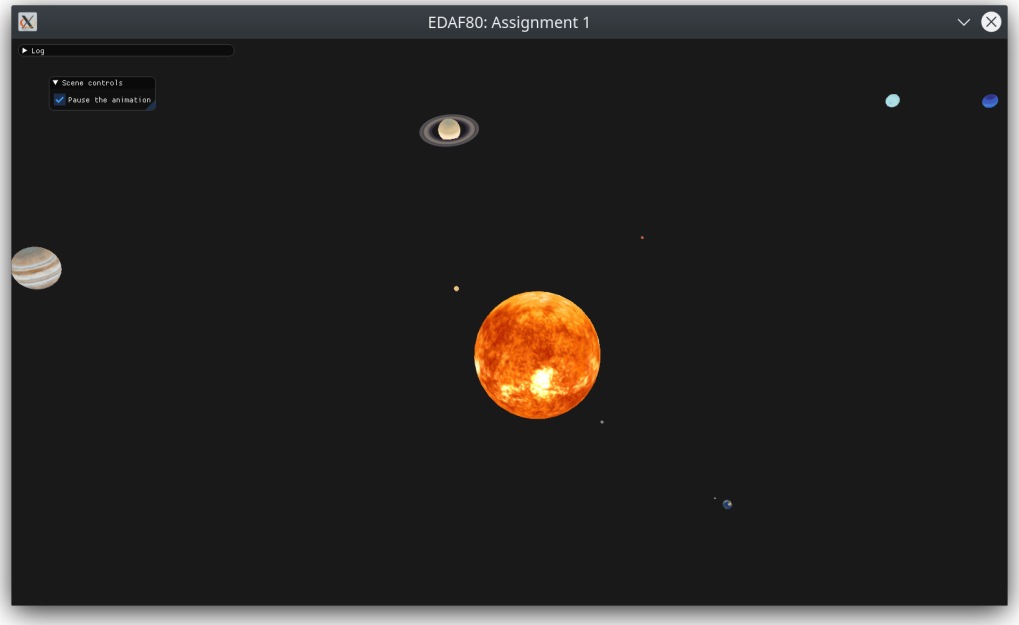

Figure 1: The end result once you have completed everything.

In this assignment you will be introduced to various transformations, as well as to the basics of hierarchical transformations through the use of a scene graph. Scene graphs let you easily express spatial relationship between the different objects in your virtual scene. For example if you are using a scene graph and setting up a kitchen scene, you might say "I am placing the plate *on the table*": you express the position of the plate as being *relative* to another object, here the table. That way if you decide to move the table to some other place in the kitchen, all the plates and glasses on it will follow, rather than stay behind and float mid-air, or forcing you to manually move all of those objects.

1 Brief overview of the framework

If you look (using a file explorer, not from within the IDE) at the content of the folder you downloaded or cloned from GitHub, you can see

- a `README.rst` file describing some basic information about the project like which libraries are used, the licence under which the code is released
- a `BUILD.rst` file explaining the softwared needed and how to build the different assignments; if you have not looked at it and set everything up yet, you should do so now before progressing further with this assignment
- a `res` folder containing all the resources (images, 3D models, etc.) used in the course
- a `src` folder for the C++ code
- and a `shaders` folder for the GLSL code that will run on the GPU (you can ignore this one until assignment 3)

The other files and folders can be ignored.

Inside `src`, you can further see

- a `core` folder which contains shared code between the two courses and is the core of the framework
- a `EDAF80` folder in which you will be doing all your modifications (along with `shaders‣EDAF80` starting with assignment 3)
- and finally a `EDAN35` folder (which can be ignored as only used for the EDAN35 course)

The documentation of the various classes and functions used in the framework can be found here. They will be presented as needed in the assignments.

2 Tips while working on the assignment

It can be hard to know if your transforms are correct, especially when they consist of multiple matrices. Start by overwriting the world matrix by each of those matrices, one at a time, to verify the effect of that single matrix. From the GUI, you can check the "Show basis" option to have an orthonormal basis drawn on screen for each celestial body, and transformed by the world matrix of that celestial body. The red arrow represents the x-axis, the green one to the y-axis, and the blue one to the z-axis. You can render additional bases by calling the `renderBasis()`[1] function with different matrices, for example if you would like to compare the effect of the world matrix versus only the orbit transform.

If you need to alter the speed at which planets are animated, you can use two different elements in the GUI:

"Pause the animation" will freeze all celestial bodies in place to allow you to get inspect them more easily

[1] `void bonobo::renderBasis(float thickness_scale, float length_scale, glm::mat4 const& view_projection, glm::mat4 const& world = glm::mat4(1.0f))`

"Time scale" will reduce or increase the speed at which time flows if you want to better observe how a planet spins, or create a timelapse effect.

The available range when dragging the slider is from 0.1× to 10×, but you can also enter custom values outside of that range by double-clicking (or control-clicking) on the value itself to get an input field.

3 Transforming an object

The `CelestialBody` is already capable of rendering itself, but it is time to add some transform functionalities, like being able to scale the celestial body and having it spin.

3.1 Scaling

We will start with allowing custom scales to be applied to a celestial body.

Exercise 1:

Let us first look at the `CelestialBody::render()` function found in `src▸EDAF80▸CelestialBody.cpp`.

1. Compute the scaling matrix $\mathcal{S}$ using GLM's `scale()`[2] function, and the class attribute `_body.scale`. The first argument should be an identity matrix, created using `glm::mat4(1.0f)`.

   This scale does not apply to its potential children.

2. Overwrite the `world` matrix with the matrix you just computed.

The rendering of a celestial body should now take its scaling into account. Go back to `src▸EDAF80▸assignment1.cpp` to test the new behaviour.

3. Compile and run the code without any additional changes. The Earth should have the same shape as before and look similar to Figure 2.

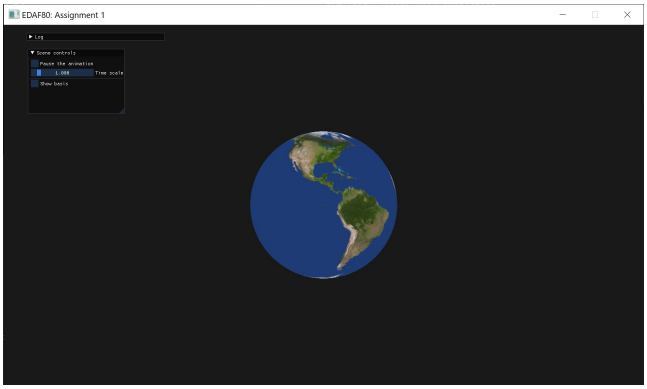

Figure 2: The Earth, with a uniform scaling of 1; only the matrix $\mathcal{S}$ is used for rendering.

4. Now scale the Earth by $\begin{pmatrix}1 & 0.2 & 0.2\end{pmatrix}$, by using its `set_scale()`[3] method. The Earth should now be closer to a rugby ball than to a sphere, and match Figure 3.

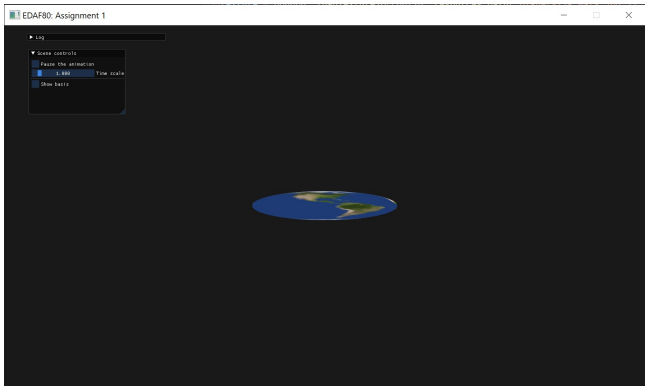

Figure 3: The Earth, with a non-uniform scaling; only the matrix $\mathcal{S}$ is used for rendering.

If your results are similar to the reference, congratulations on applying your first transformation! Keep the scaling applied to the Earth, as it will stay useful.

3.2 Spinning

An object spins by rotating around an axis passing through the object's centre of gravity. The spin of a celestial body is parametrised by the following attributes:

`_body.spin.axial_tilt` [rad] how much is the spin plane (i.e. the plane perpendicular to the spin axis) tilted relative to its orbit plane.

`_body.spin.speed` [rad/s] how fast it spins;

`_body.spin.rotation_angle` [rad] how much has it already rotated.

The spin transformation will be decomposed into two different rotations. A first one, $\mathcal{R}_{1,s}$, which represents the celestial body spinning around the y-axis. The second one, $\mathcal{R}_{2,s}$, will tilt the spin plane by the specified axial tilt around the z-axis.

Both rotation matrices should be computed using GLM's `rotate()`[4] function. The first argument will once again be an identity matrix.

Exercise 2:

We will go back to the `CelestialBody::render()` function found in `src▸EDAF80▸CelestialBody.cpp`.

1. Compute the rotation matrix $\mathcal{R}_{1,s}$; you can temporarily overwrite the `world` matrix with it. This should be performed *after* the `_body.spin.rotation_angle` attribute has been updated.

2. Compile and run the code; if everything worked, the Earth should have spinned around the y-axis by a certain amount and look similar to Figure 4; you can compare to Figure 2 to see the impact of the rotation more easily.

[2] `glm::mat4 glm::scale(glm::mat4 const& matrix, glm::vec3 const& scale);`

[3] `void CelestialBody::set_scale(glm::vec3 const& scale)`

[4] `glm::mat4 glm::rotate(glm::mat4 const& matrix, float angle, glm::vec3 const& axis);`

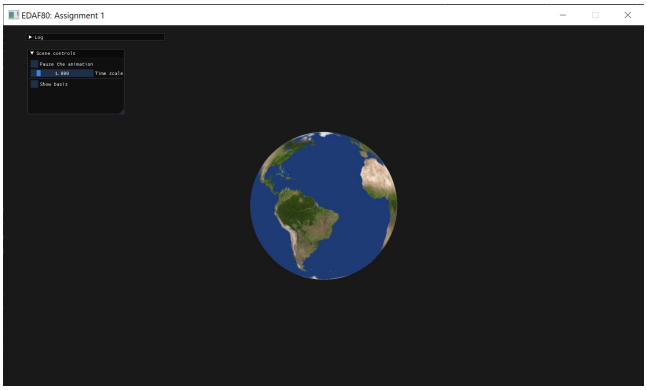

Figure 4: Putting a spin on the Earth; only the matrix $\mathcal{R}_{1,s}$ is used for rendering.

3. Compute the rotation matrix $\mathcal{R}_{2,s}$; you can temporarily overwrite the `world` matrix with it. If you compile and run the code, the Earth should have rotated a bit around the z-axis (compared to Figure 2) and be similar to Figure 5.

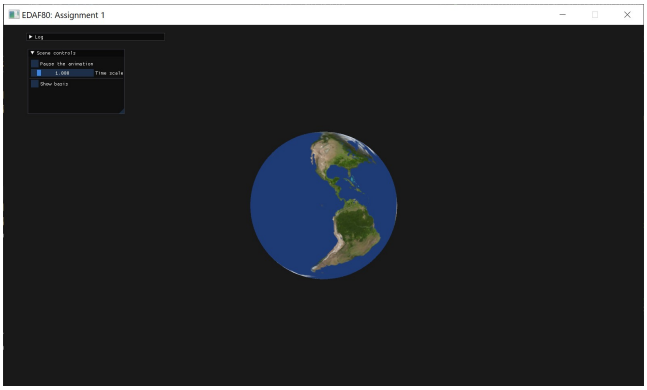

Figure 5: Visualising the axial tilt transform; only the matrix $\mathcal{R}_{2,s}$ is used for rendering.

4. Set the `world` matrix to a combination of $\mathcal{R}_{1,s}$ and $\mathcal{R}_{2,s}$; think about which of the two matrices should be applied first to the celestial body. You should end up with an Earth looking like the one in Figure 6.

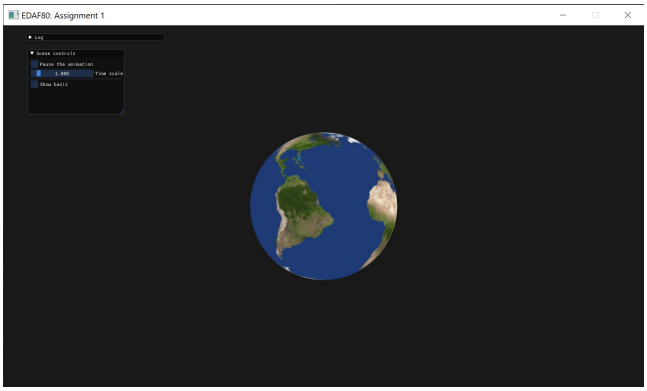

Figure 6: The full spin transform; still static; only the matrices $\mathcal{R}_{1,s}$ and $\mathcal{R}_{2,s}$ are used for rendering.

5. In the code, *before* $\mathcal{R}_{1,s}$ is computed, update the spin angle by how much the celestial body rotated since last frame. Since the rotational speed for the spin is defined in rad/s and not rad/μs, `elapsed_time_s` should be used rather than `elapsed_time`.
6. Combine the scaling matrix with the two matrices responsible for the spin; it should make it easier to confirm that the Earth is indeed spinning around a tilted axis.

3.3 Adding orbits

With the possibility of scaling or making a celestial body spin, the main missing part, in terms of individual transformations, is giving it an orbit.

The difference compared to the spin, is that an orbit is a rotation around a point in space located outside of the celestial body. By default, the centre in object space usually is located at the centre of the object, so applying a rotation will result in the object spinning. By first moving the centre of the object away from the centre of the local space, the rotation will still be around the centre of the local space but it no longer is the centre of the object: the object will start describing an orbit.

To move the object away, we will be translating in a certain direction. The translation matrix can be computed using GLM's `translate()`[5] function. The first argument remains an identity matrix.

Exercise 3:

1. Compute the translation matrix, $\mathcal{T}_o$, which places the celestial body onto its orbit, i.e. offset the celestial body by its orbit radius along the x-axis. Temporarily overwrite `world` to verify that the translation works and you obtain something similar to Figure 7.

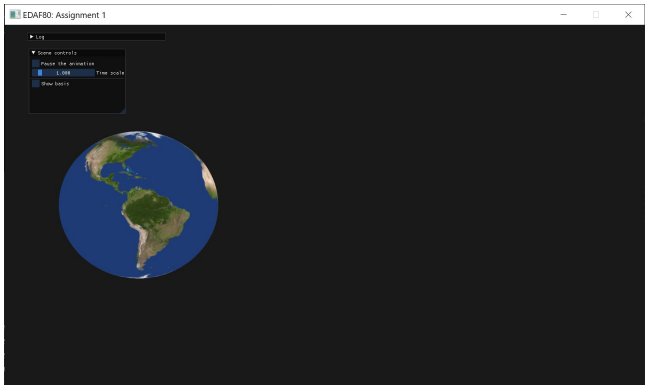

Figure 7: Moving the Earth to another place; only the matrix $\mathcal{T}_o$ is used for rendering.

2. Combine $\mathcal{T}_o$ with the scaling and spin matrices, and ensure that you get the Earth spinning at a certain distance from the centre of the screen. The Earth should *not* be describing an orbit just yet.
3. Update the orbit angle every frame, similar to how you updated the spin angle in the previous section, and compute the rotation matrix, $\mathcal{R}_o$, that will make the celestial body rotate around its orbit centre in the x, z plane. If you would like to check the validity of $\mathcal{R}_{1,o}$, overwrite `world` with it; since you only have a rotation being applied, you will end up with the Earth spinning around its y-axis.
4. Combine $\mathcal{T}_o$ and $\mathcal{R}_{1,o}$, and ensure that the Earth describes an orbit around the centre of the screen. It might look like the Earth is also spinning, however it is not and comes from the orbiting computations making the Earth keeps its orientation relative to the centre of its orbit. Indeed you can observe that, in this case,

[5] `glm::mat4 glm::translate(glm::mat4 const& matrix, glm::vec3 const& vector);`

Africa and Europe are always facing towards the centre of the orbit, no matter what.

5. Compute the matrix tilting the orbit plane around the z-axis, $\mathcal{R}_{2,o}$, similar to the computation of the matrix tilting the spin plane.
6. Combine $\mathcal{R}_{2,o}$ with $\mathcal{T}_o$ and $\mathcal{R}_{1,o}$ to get the full orbit transformation.

   Once this works, add the scaling and spin back into the `world` matrix computation.

4 Implementing a scene graph

If you look inside the rendering loop (it corresponds to the `while (!glfwWindowShouldClose(window)) { ... }` block found in `src‣EDAF80‣assignment1.cpp`), you will see that we currently do not traverse any scene graph but are instead explicitly rendering the Earth node. We could call the `CelestialBody::render()` method on each and every celestial body, but it will become tedious having to call the render function every time you add a new celestial body (and it's possible to forget doing it in the first place). So instead, we will write some code to automatically render all the nodes within our scene graph.

4.1 Establishing a simple parent-child relationship

Before looking at traversing the whole scene graph, we will start by making sure that the transform of a parent is correctly applied to its children. To that end, the current setup of only having the Earth will be extended by adding the Moon and designing it as a child of the Earth by doing `earth.add_child(&moon)`.

Exercise 4:

1. In the `CelestialBody::render()` method, take into account the previously-ignored matrix `parent_transform` when computing `world`; think about which transformations should be applied first to an object: its own transformations, or its parent ones. The Earth should still be spinning and have a weird scale, but instead of orbiting towards the left of the screen, it should now be orbiting a point located more on the right of the screen.
2. Compute the matrix which should be applied to all children of the current celestial body, and return it instead of `parent_transform`. It should be similar to the computation of the `world` matrix, however the scale and spin rotation should *not* affect the children and therefore be ignored in the computation; the spin tilt *is* still taken into account and will affect the children.
3. Uncomment the call to render the Moon in `src‣EDAF80‣assignment1.cpp`, and forward as third argument, the matrix returned by the rendering of the Earth. The Moon should now be orbiting around the Earth, no matter where the Earth goes.
4. You can now remove the weird scaling from the Earth.

4.2 Traversing the scene graph

Since parents keep track of who their children are (but not the other way round), the graph will be traversed from top to bottom. This could be done in a depth-first or breadth-first search, but for simplicity we will do a depth-first search traversal (see https://en.wikipedia.org/wiki/Depth-first_search. Depth-first means that after we are done processing the current node, we will start processing its children *before* processing its siblings; this is done iteratively until no children are left. Once we end on a node with no children, we backtrack and go on processing the last child we saw but left for later, and go on from there, until we have visited all the nodes in the graph.

To help with implementing this depth-first search traversal, you are encouraged to use the `std::stack` class from the C++ standard library (see https://en.cppreference.com/w/cpp/container/stack; do not forget to add `#include <stack>`). You can use the `push(object)` method to place `object` on top of the stack, the `object& top()` method to get the object located at the top and `pop()` to remove it, and finally, the `bool empty()` method to know whether the stack is empty or not.

You can also implement the depth-first search traversal recursively; the information below applies for the iterative approach.

Exercise 5:

1. Create a stack containing `CelestialBodyRef`, which will keep track of unprocessed celestial bodies along with their parent transform.
2. Initialise the stack with the root node of the graph: the Earth. As the Earth does not have a parent at the moment, use the same translation matrix that is passed as third argument to the explicit rendering of the Earth.

   `CelestialBodyRef` is a structure made of two members: a pointer, named `body`, pointing to an existing `CelestialBody`, and a matrix named `parent_transform` which contains the parent transform for the celestial body referenced by `body`.
3. Remove the explicit rendering of the the Earth and the Moon.
4. Implement the depth-first search traversal of the graph, starting from the root node. Do not forget to forward the appropriate parent transform for the rendering.
5. Check that starting the traversal from the root node results in both the Earth and the Moon being rendered, similarly to before.

5 Building the Solar System

You now have all the pieces necessary for building the whole Solar System!

Exercise 6:

1. Start by changing the scale and orbit values for the Earth and Moon to instead use those defined in `<planet>_scale` and `<planet>_orbit`.
2. Add all the remaining planets in the Solar System, as well as the Sun which will be used as the new root node of the scene graph (all celestial bodies should be children of the Sun). All the needed constants and textures are available in variables named following the pattern `<planet>_scale`, `<planet>_spin`, `<planet>_orbit`, `<planet>_texture`, but you are free to use your own constants if you prefer.

   To get a better overview of all the celestial bodies, we recommend changing the camera translation to $\begin{pmatrix}0 & 4 & 20\end{pmatrix}$; look for `camera.mWorld.SetTranslate()` towards the beginning of `src ‣ EDAF80 ‣ assignment1.cpp`. Note that you can make the window fullscreen by pressing F11 (and use the same key to go back to windowed); you can find additional controls in Table 1.
3. Once you are done, you should have something resembling Figure 1.

6 Suggestions and things to ponder

Exercise 7:
Adding rings to a celestial body:

Table 1: Various controls when running an assignment."Reload the shaders" is not available in assignments 1 and 2 of EDAF80, while "Toggle fullscreen mode" is missing from assignment 2 of EDAN35.

Action	Shortcut
Move forward	W
Move backward	S
Strafe to the left	A
Strafe to the right	D
Move downward	Q
Move upward	E
"Walk" modifier	⇧
"Sprint" modifier	Ctrl
Reload the shaders	R
Hide the whole UI	F2
Hide the log UI	F3
Toggle fullscreen mode	F11

1. Update the `CelestialBody::render()` method to render the rings. This is achieved by calling the `Node::render()` method on the rings node.

   Rotate the rings by 90° around the x-axis, *after* having scaled them, to bring them into the celestial's body equatorial plane. The scaling factors for the rings are given in the x, y-plane, the plane in which the ring shape is defined.

   Since we will be considering the rings to be a child of the celestial body, which transformations should be applied to the rings?

2. Check that it works by adding rings to Saturn using the `set_ring()`[6] method of the `CelestialBody` class.

   The ring shape has already been created can be found in `saturn_ring_shape`. The program is stored in the variable `celestial_ring_shader`, while the diffuse texture is found in the variable `saturn_ring_texture`, and the scale in `saturn_ring_scale`.

Exercise 8:
Right now, you can see that the tilt of a celestial body is always facing towards the centre of its orbit. This is valid for the Moon, but not so much for the Earth as our existing seasons would not exist as a result. How would you make the tilt rotation independent?

Exercise 9:
How would you create an "interplanetary tour" where the camera would follow one of the animated objects? This would require working out the location of the camera first, then applying it to the camera's position.

A Framework controls

The framework uses standard key bindings for movement, such as W, A, S, and D. But there are also custom key bindings for moving up and down, as well as controlling the UI. All those key bindings are listed in Table 1.

There is only one action currently bound to the mouse, and that is rotating the camera. To do so, move the mouse while holding the left mouse button.

GUI elements can be toggled being a collapsed and expanded state by double clicking on their title bar. And they can be moved around the window by dragging their title bar wherever desired (within the window).

B IDE key bindings

To help with getting certain tasks done more efficiently, Table 2 lists key bindings of different IDEs for several common actions.

[6] `void CelestialBody::set_ring(bonobo::mesh_data const& shape, GLuint const* program, GLuint diffuse_texture_id, glm::vec2 const& scale)`

Table 2: Various keyboard shortcuts for Visual Studio 2019 and 2017, and Xcode.

Action	Shortcut	
	Visual Studio	Xcode
Build	Ctrl + B	⌘ + B
Run (with the debugger)	F5	⌘ + R
Run (without the debugger)	Ctrl + F5	
Toggle breakpoint at current line	F9	⌘ + \
Stop debugging	⇧ + F5	⌘ + .
Continue (while in break mode)	F5	ctrl + ⌘ + Y
Step Over (while in break mode)	F10	F6
Step Into (while in break mode)	F11	F7
Step Out (while in break mode)	⇧ + F11	F8
Comment selection	Ctrl + K, Ctrl + C	⌘ + /
Uncomment selection	Ctrl + K, Ctrl + U	⌘ + /
Delete entire row	Ctrl + X	