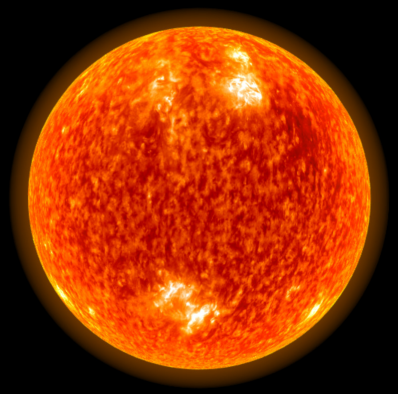
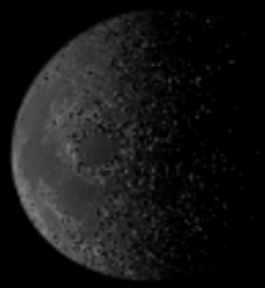
香港科技大學
THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

CSIT5400 Computer Graphics

# Assignment 2 –
# The Earth, the Moon and the Sun



# The Earth, the Moon and the Sun

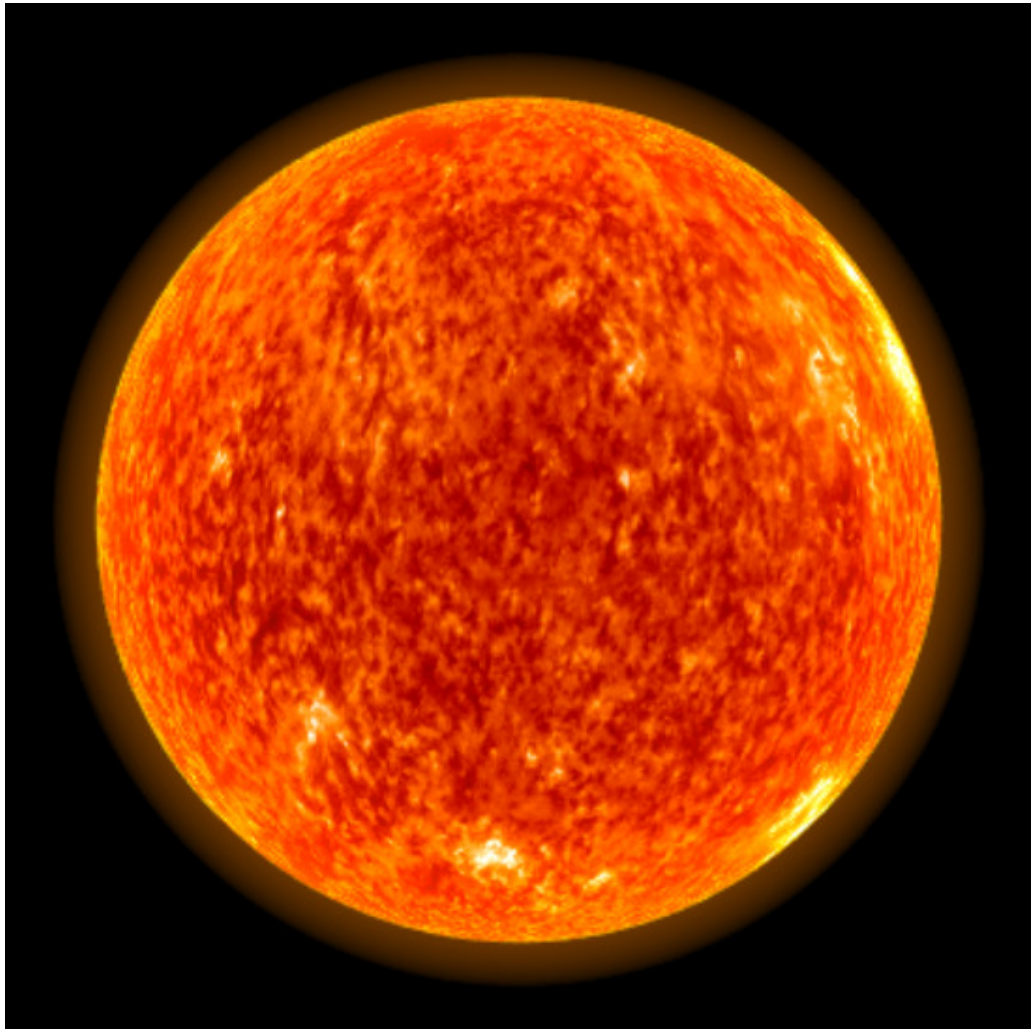


# The Earth, the Moon and the Sun

- In this assignment, you will render the Earth, the Moon and the Sun using:
  - Texture mapping techniques
  - Hierarchical modeling
- You will start from a plain sphere and gradually make your system look real
- We will simply call them ‘EMS’ in the rest of this discussion to save some space

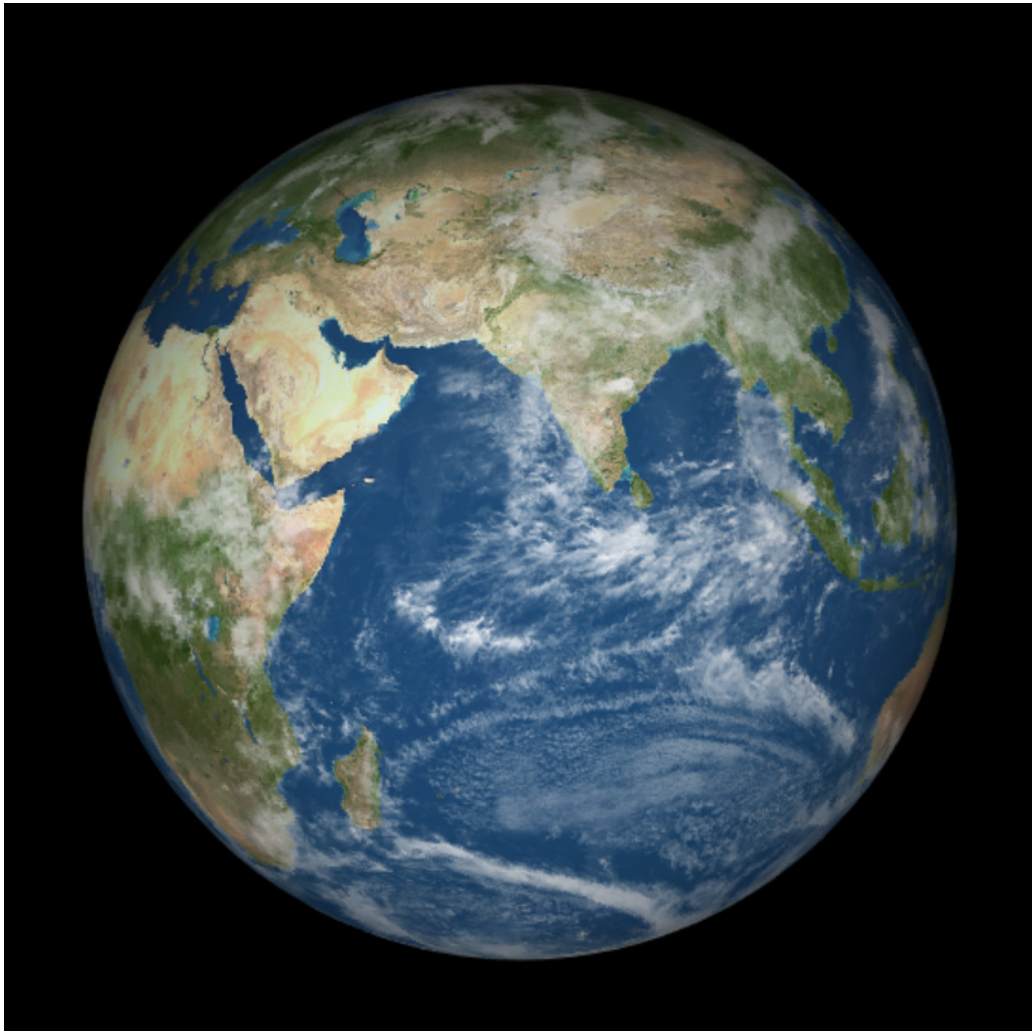
# The Sun

- Here is the display of the Sun
- The Sun is at the centre of our EMS system
- It is a light emitter so no matter where you look at it, it is always bright



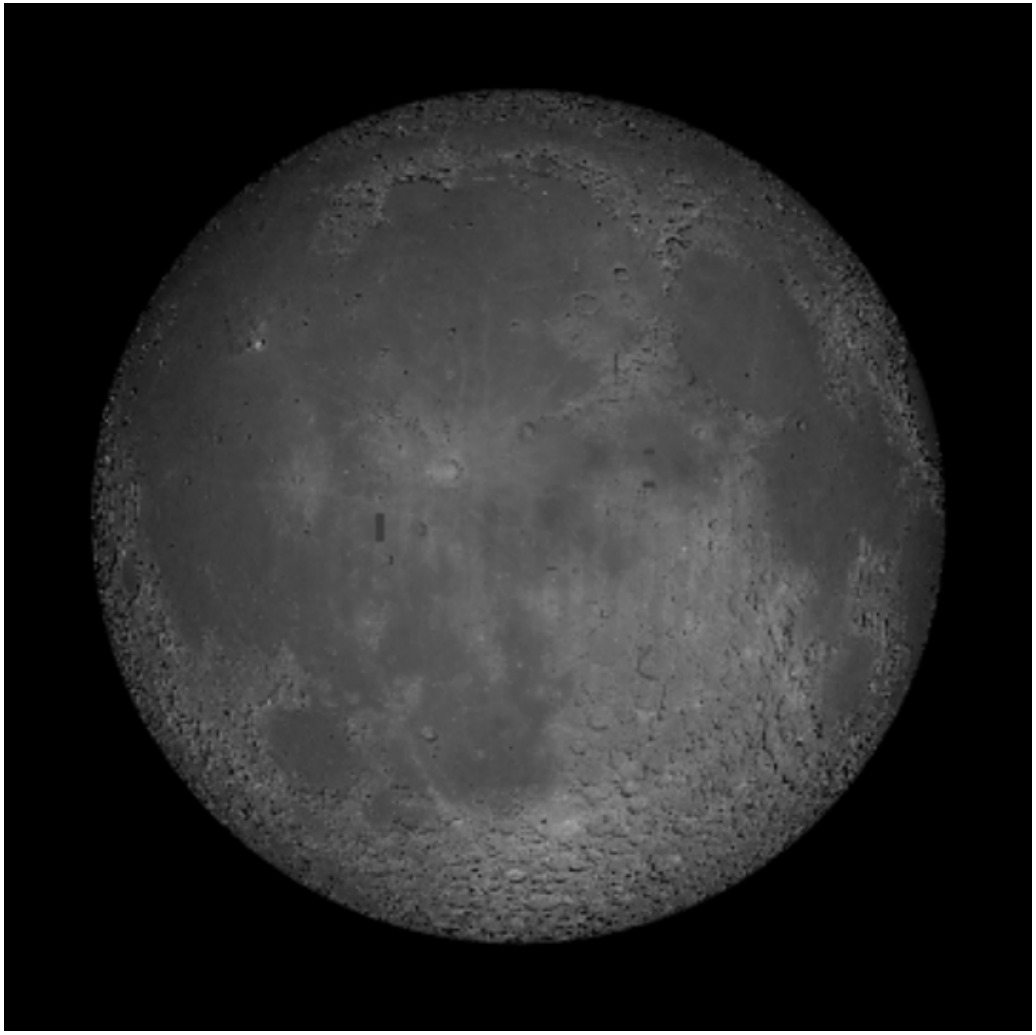
# The Earth

- Here is the display of the Earth
- The Earth is slightly away from the Sun
- Only the side that is facing the Sun is illuminated (day); the other side is at night



# The Moon

- Here is display of the Moon
- The Moon is very close to the Earth
- Although it rotates, i.e. orbits, around the Earth, the side facing the Earth is always the same

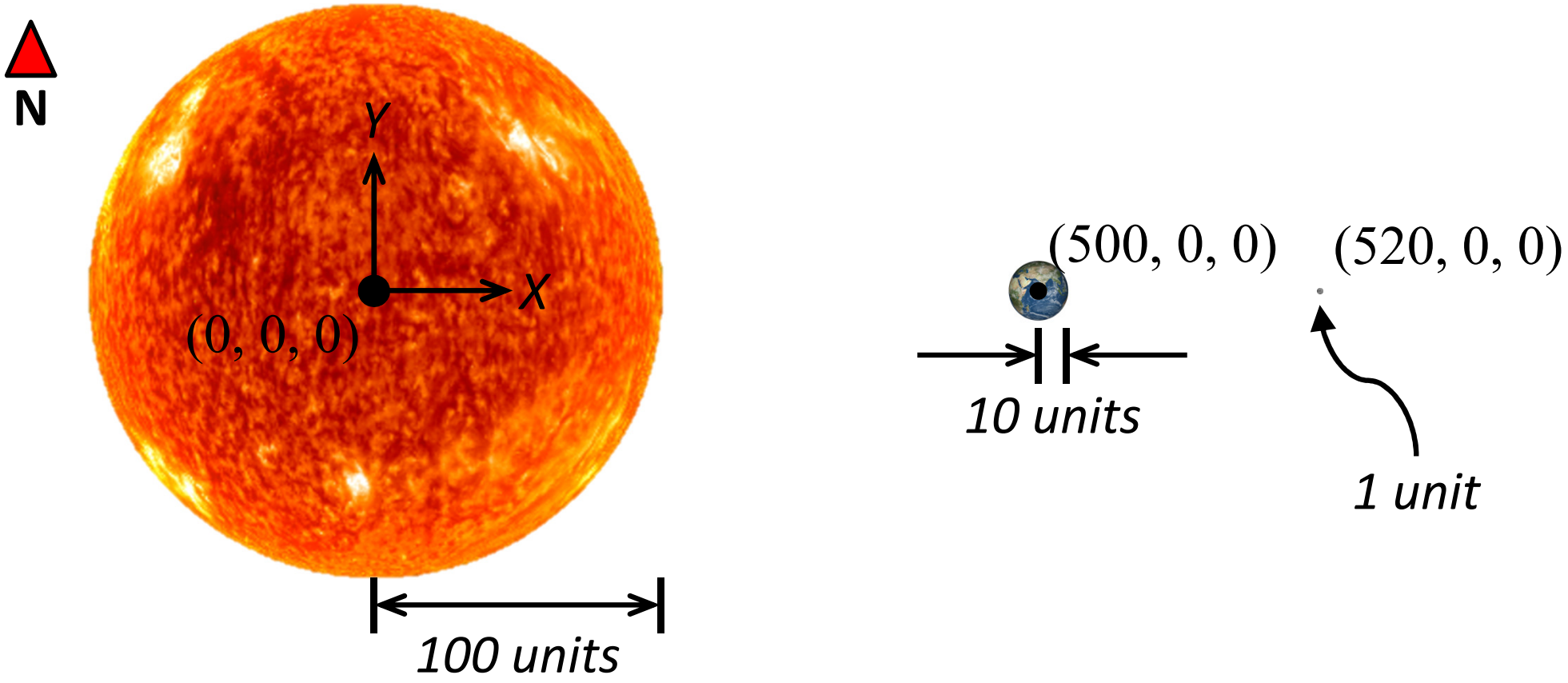


# Locations and Sizes

- Here are the basic settings of the EMS system:
  - The radiuses of the Sun, the Earth and the Moon are 100 units, 10 units and 1 units respectively
  - The location of the Sun is at $(0, 0, 0)$
  - The location of the Earth is initially at $(500, 0, 0)$, i.e. 500 units away from the Sun
  - The location of the Moon is initially at $(520, 0, 0)$, i.e. 20 units away from the Earth
  - The Y axis points towards the north

# Initial Positions and Sizes

- This shows the initial positions of the EMS (they are not drawn to scale)



# Interactions Between EMS

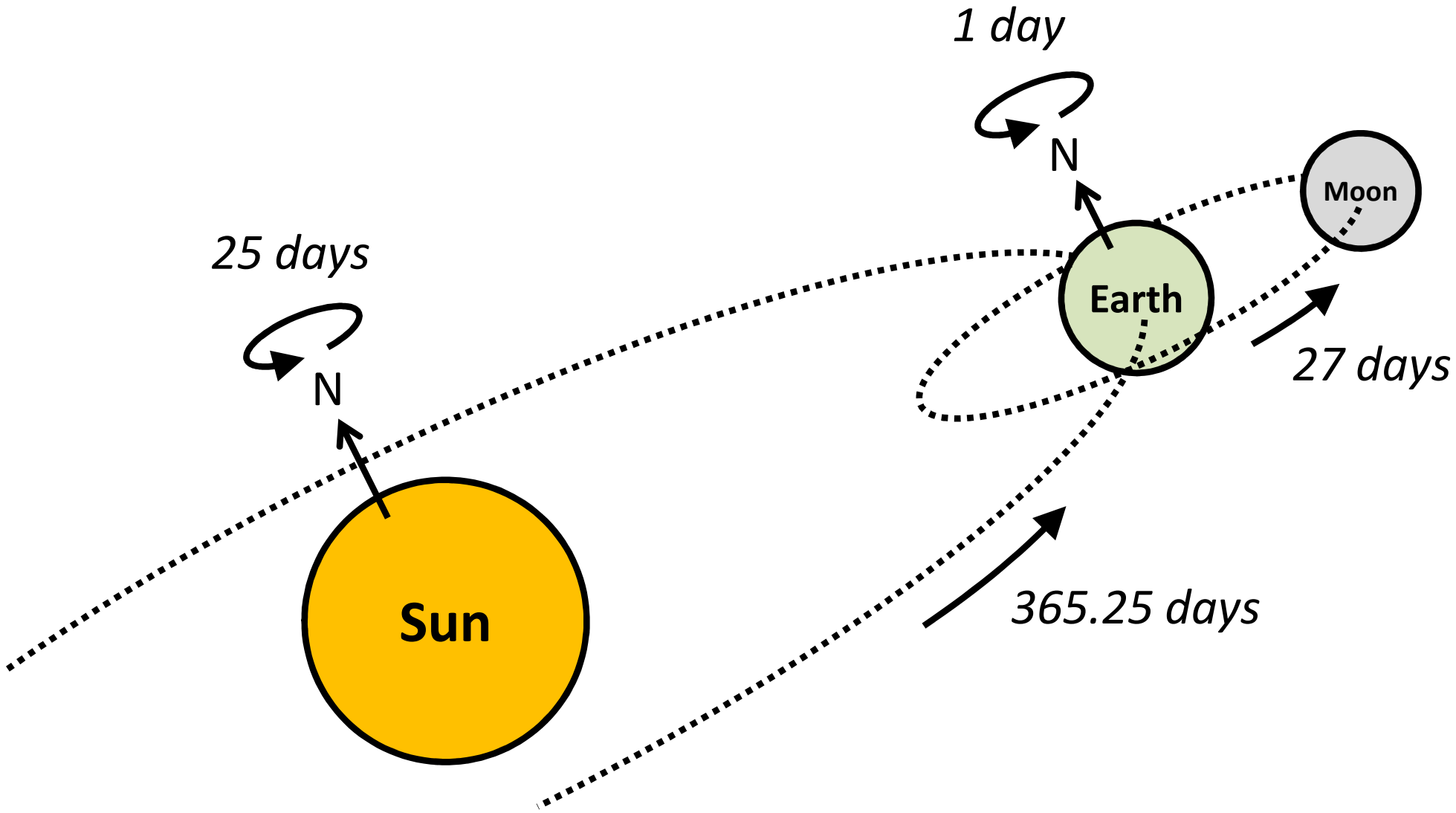
- You probably know that most celestial objects rotate around their axes of rotation
- You also likely know that some celestial objects orbit around other objects that are more massive than themselves
- Each of the EMS also does that with the Earth orbiting the Sun and the Moon orbiting the Earth
- Their interactions are nicely summarized here:

http://kidseclipse.com/sun-earth-moon-move/

# Rotation and Orbit Timing

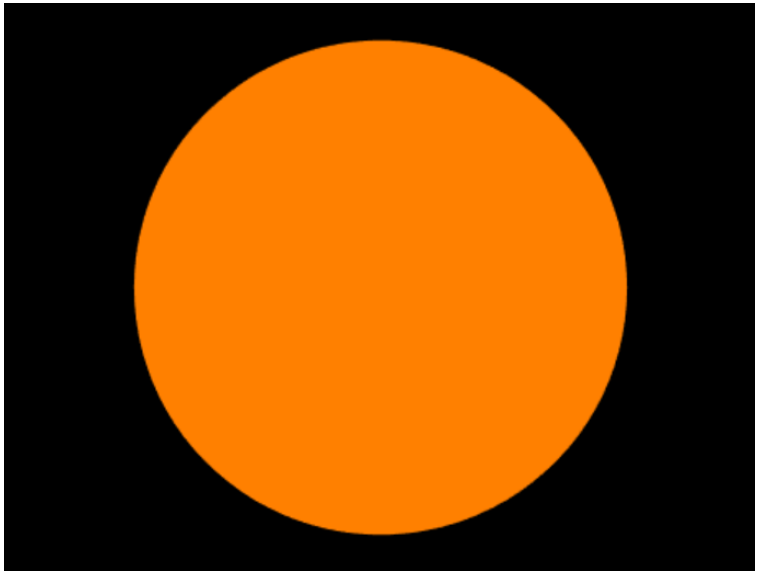
- The Sun completes one period of rotation (from west to east) around its axis of rotation in **25 days**
- The Earth completes one period of rotation (from west to east) around its axis of rotation in **1 day**
- The Earth completes one orbit around the Sun in **365.25 days**
- The Moon completes one orbit around the Earth in **27 days**
- We will use these numbers even though they are approximations

# Rotation and Orbit Directions



# Starting Project

- You will be given a starting project that displays a plain orange sphere at the centre of the canvas



- Similar to previous examples, you can rotate and zoom the sphere around using the mouse, combined with the ctrl key (no panning in the assignment)
- The project uses Phong shading and therefore lighting calculation is done in the fragment shader

# The Starting Code and GLSL

- Some comments have been added to the starting code so that you should be able to find the locations where you need to add the code for the assignment
- Part of the code you will write this time is GLSL code
- Working in GLSL is hugely different to JavaScript because it is not easy to debug in GLSL
- We will briefly talk about debugging in GLSL and some reminders about using GLSL

# Creating the Sphere

- If you look at the code, you will see the project uses slightly different code to create the sphere buffer, as shown below:

```
var arrays =
    primitives.createSphereVertices(1, 50, 50);
createSphereTangents(arrays, 50, 50);
sphere = twgl.createBufferInfoFromArrays(gl, arrays);
```

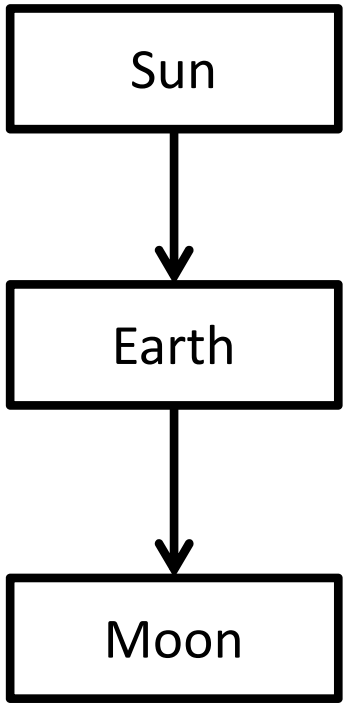
- Rather than directly creating a sphere using TWGL, the code also generates tangents for the vertices, which will be used later in the assignment

# Your Tasks

- Given the starting code, you will improve it by:
  1. Position the EMS using hierarchical modelling
  2. Animate the EMS interaction within the system
  3. Position the camera in front of the Sun, the Earth or the Moon
  4. Use the Sun as the light source
  5. Render the Sun appropriately
  6. Render the Earth appropriately
  7. Render the Moon appropriately

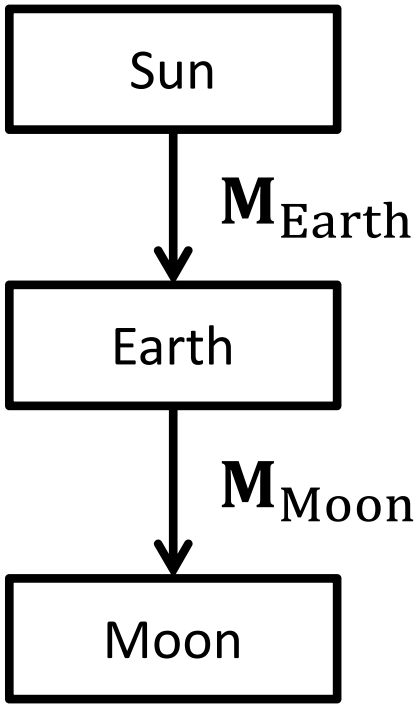
# Task 1. Hierarchical Modeling

- To simulate the EMS interaction, it is important to use a hierarchical structure
- The structure has three components, the Sun, the Earth and the Moon, where the Sun is at the top and the Moon is at the bottom
- This is similar to the robot arm structure shown in the lectures



# Working Out the Transformations

- The starting code has already pointed out the locations that you need to work on for the structure
- You need to distinguish between transformations that you need to progress through the structure and the ones that you need for model transformation



# Using the Structure

- If you set them up correctly, the positioning for the EMS will look like this:

# Task 2. Animating the Interactions

- After fixing their structure and initial positions, you can work on their interactions
- Within the system, all interactions are achieved by rotational transformations
- These transformations will have to be inserted into appropriate places:
  - Self rotation would be part of the model transformation
  - Orbital rotation would be in the hierarchical structure

# Timing of the Animation

- So the Earth completes one period of rotation in 1 day and it would be too slow to show for your program
- In the assignment, the content is updated 60 times a second (= 60 frames per second)
  - This is controlled by `requestAnimationFrame()`
- Then, you will use 60 frames in the assignment to represent 1 day in the system
- In other words, the Earth completes one period of rotation in exactly 1 second inside the assignment

# Finding the Rotation Angle

- For example, if the Earth is to rotate one period in 1 second, the angle of rotation will be adjusted, for each frame, like this:

$$\text{angle} = \text{angle} + 1 \,/\, 60 \times 2\pi$$

  - In the above equation, the angle value is expressed in radians
- You can similarly figure out the angle of rotation for the rest of the interactions

# Controlling the Animation Speed

- The animation speed can be adjusted by pressing the '+' key or '-' key
- If you look at the code, a variable `animateSpeed` has been created for you to control the animation
- The variable is a ratio (initially at 1.0) that can be used to adjust the angles for applying your rotations, e.g.:

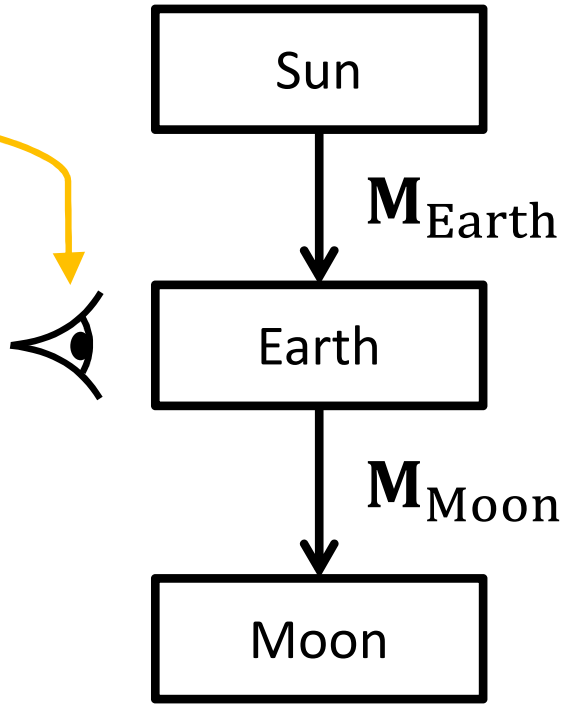
$$\text{angle} = \text{angle} + 1 / 60 \times 2\pi \times \texttt{animateSpeed}$$

# Task 3. Positioning the Camera

- At the moment, the camera is always set in front of the Sun (see the use of `m4.lookAt()`) so that it is not easy to find the Earth and the Moon
- To improve the experience, you can put the camera in different places
- Based on the value of the `cameraPos` variable, you need to put the camera in front of either the Sun (=1), the Earth (=2) or the Moon (=3)

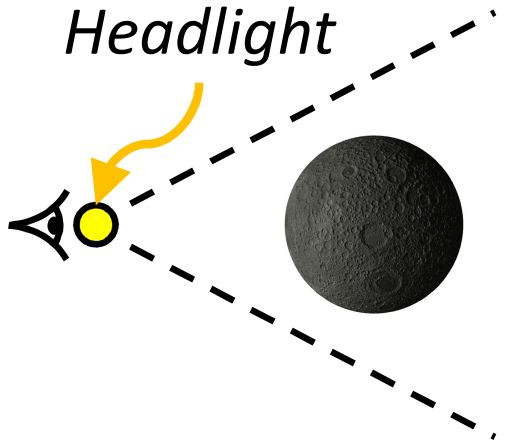
# Putting the Camera for the Earth

- Let's say you want to put the camera in front of the Earth
- To do that, you need to know where the Earth is
- This can be done by multiplying $\mathbf{M}_{\text{Earth}}$ to $(0, 0, 0)$
- Similarly, if you put the camera in front of the Moon, you will need to multiply $(0, 0, 0)$ with $\mathbf{M}_{\text{Earth}}\mathbf{M}_{\text{Moon}}$



# Task 4. Using the Sun as Light Source

- The starting code gives you a ‘headlight’ as the light source
- The light source location is passed to the shader using this line of code:




```
uniforms.u_LightPos = [0, 0, 0];
```

- Since lighting calculation is done in the eye space, the above point `[0,0,0]` is the location of the eye
- You need to change the point so that the light source is at the Sun’s position

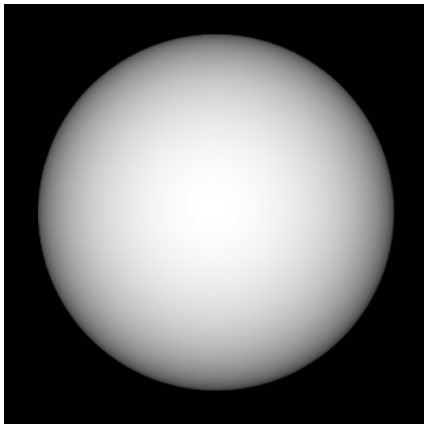
# Changing the Light Source Position

- You need to use a light source which is located at $(0, 0, 0)$ in the world space, i.e. the location of the Sun
- Before you pass the light position to the shader, you need to transform it to the eye space by the **viewing transformation**, i.e.:

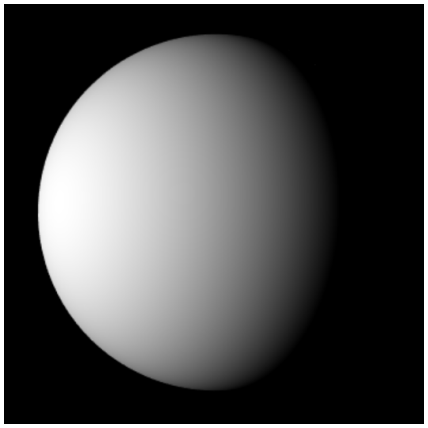
$$\mathbf{P}_{\mathrm{e}} = \mathbf{M}_{\mathrm{view}}\mathbf{P}_{\mathrm{w}}$$

*The position that will be passed to the shader*

$(0,0,0)$



*Using headlight*



*Using the Sun*

# Transforming the Light Position

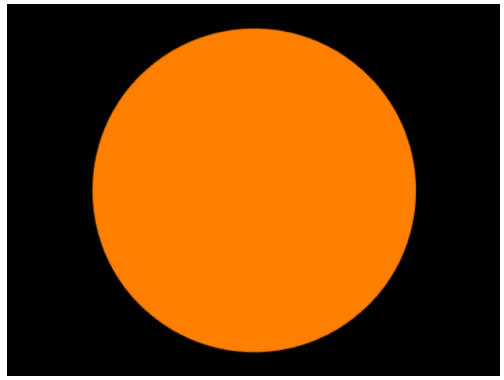
- To transform the light position using the viewing transformation, you can make use of the `m4.transformPoint` function in TWGL
- The function (pre-)multiplies a given point with a matrix and returns the result, as shown below:

`m4.transformPoint(` *matrix* `,` *point* `)`

- You can obtain the viewing transformation matrix from the starting code, i.e. `viewMatrix`

# Task 5. Rendering the Sun

- At the moment, the Sun is a simple orange circle without any shading
- You can see that it is a ‘light emitter’ because the colour given out by the sphere is a constant, i.e.:

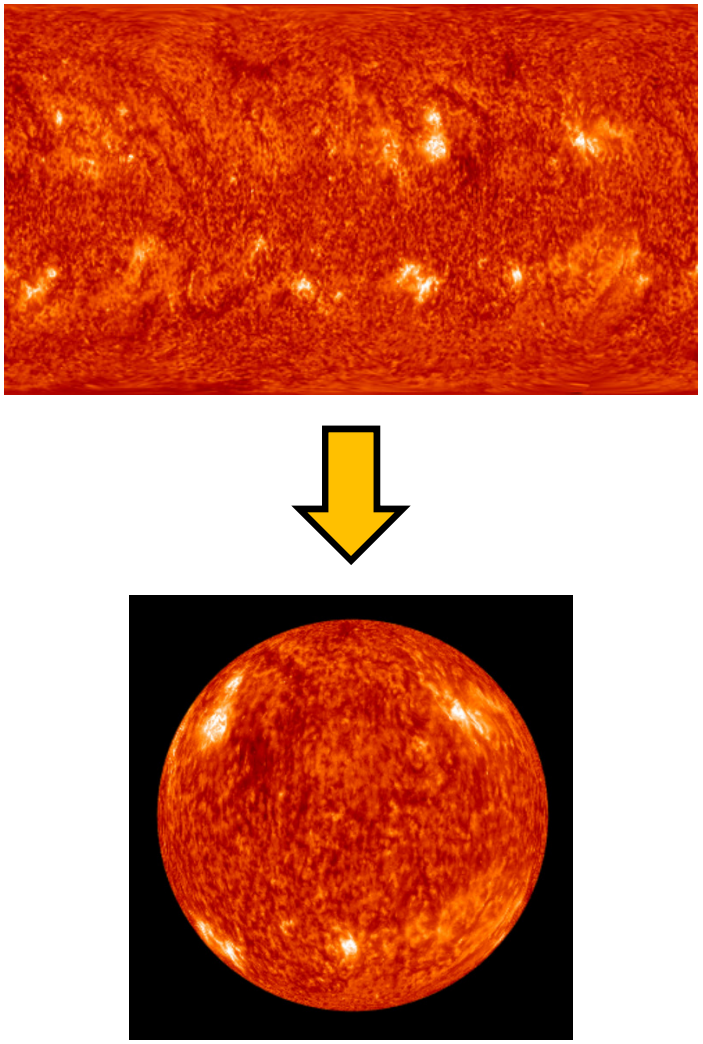


```
gl_FragColor = vec4(vec3(1, 0.5, 0), 1);
```

- To make the Sun look realistic, you need to add a texture, as well as some Sun features

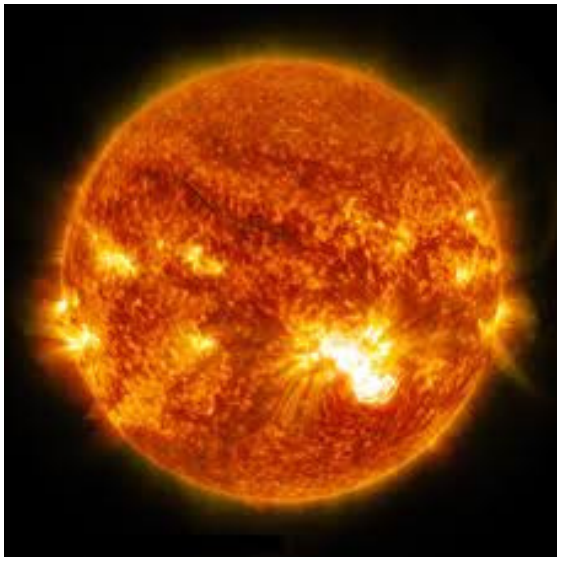
# The Sun Colour Map

- You are given a texture as the Sun's colour map
- You should have no problem applying the texture based on the texture mapping example given to you
- Since it is an emitter, you should use `vec3(1,1,1)` as the shading colour of the Sun



# Additional Sun Features

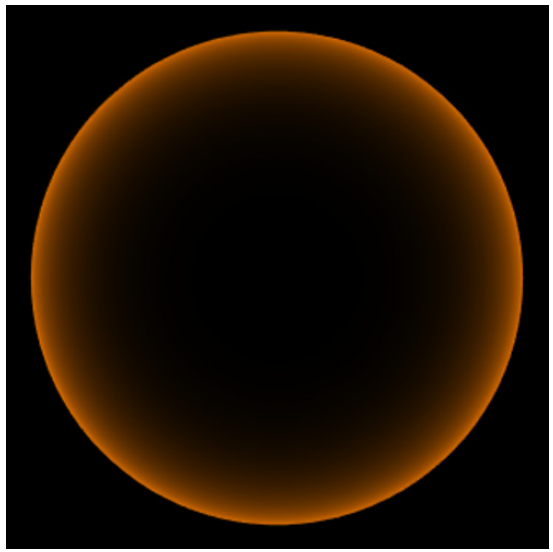
- It is a bit boring with only the colour map
- You can make it more ‘fierce’ by adding two features:
  - An inner glow, making a ‘burning’ glow inside the silhouette
  - An outer glow, putting a ‘burning’ glow extending out of the silhouette of the sphere



*A more interesting display of the Sun*

# Burning Glow Inside

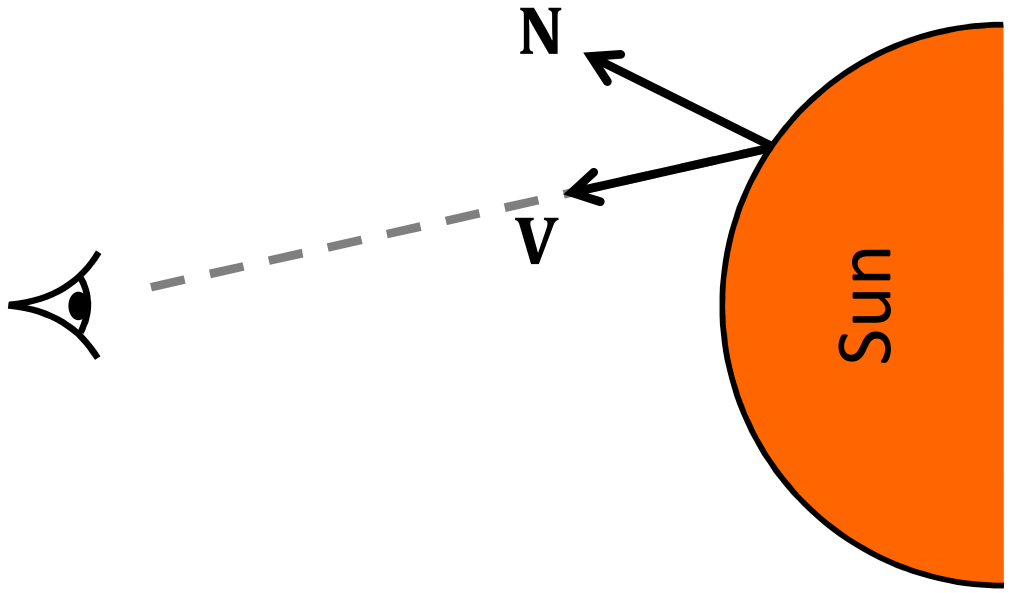
- The idea is to add a burning glow around the inside of the silhouette
- The inner glow decreases as you move inwards
- For example, the image on the right shows the orange ‘burning glow’ on the sphere, without showing any other texture or shading colour:



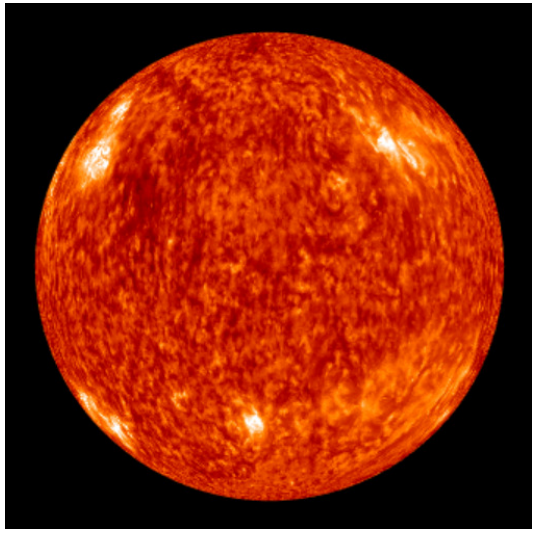
# Finding the Inner Glow

- The amount of burning glow inside the silhouette can be calculated using a simple dot product of the viewing direction and the normal
- You may also adjust the amount of glow using a power $\boldsymbol{n}$, i.e.

$$(\mathbf{V} \cdot \mathbf{N})^{\boldsymbol{n}}$$

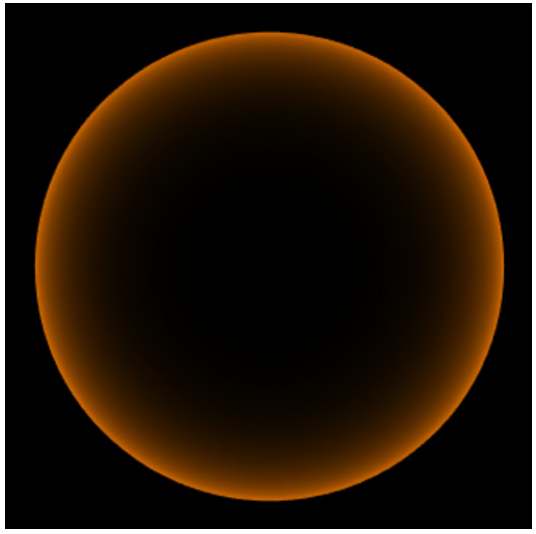


# Combining the Texture With Inner Glow



*The colour map*

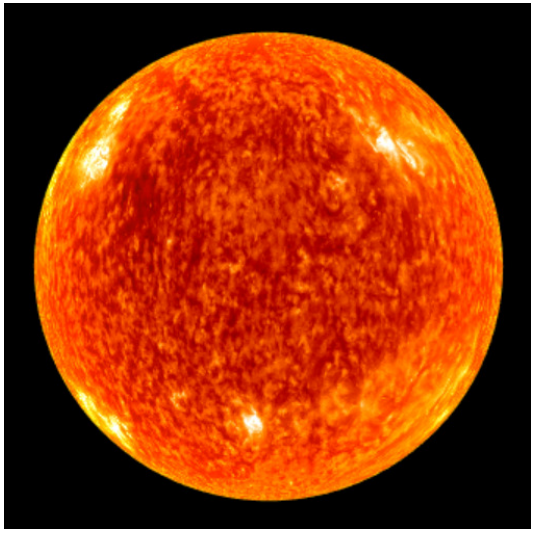
+



*The inner glow*

*(this example uses a power of 2 on the dot product)*

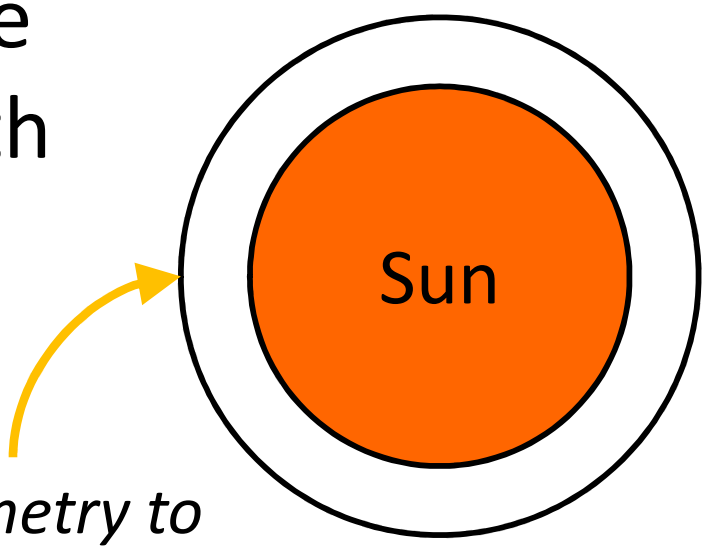
=



*The final result*

# Burning Glow Outside

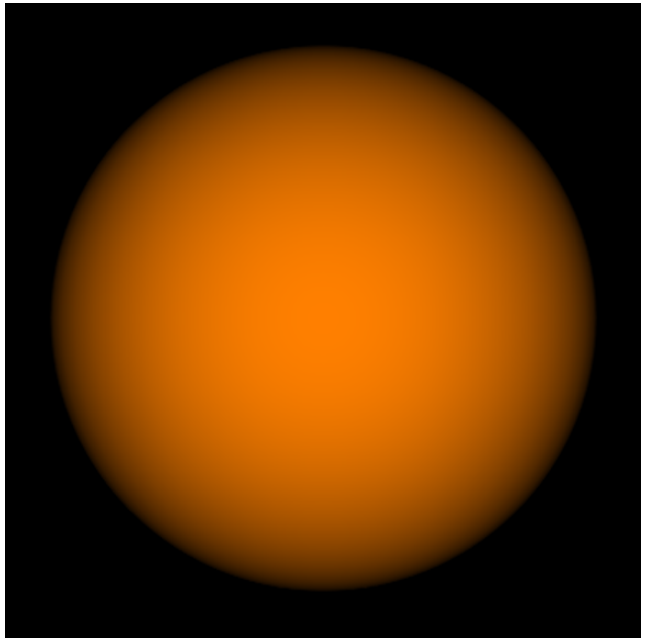
- A burning glow can be extended outside the silhouette of the Sun
- To do that, you need a new geometry
- The geometry can just be the same sphere but applied with a magnifying scaling factor, e.g. 1.2




*New geometry to draw the outer grow*

# Finding the Outer Glow

- In the exact opposite to the inner glow, the glow is the strongest at the centre and diminishes towards the silhouette
- Again, using a dot product you can produce an outer glow like the one shown on the right:



# The Outer Glow Hiding the Sun

- If you draw the Sun and the outer glow at the same time, you will not see the Sun anymore because the outer glow will always hide the Sun
- A quick trick to show the Sun is to disable the depth buffer when you draw the outer glow geometry
- If you do that, the outer glow will be drawn normally but the depth buffer is not updated
- Then, anything drawn later will overwrite the outer glow if they overlap

# Enable / Disable the Depth Buffer

- To disable the depth buffer, you can use this code:

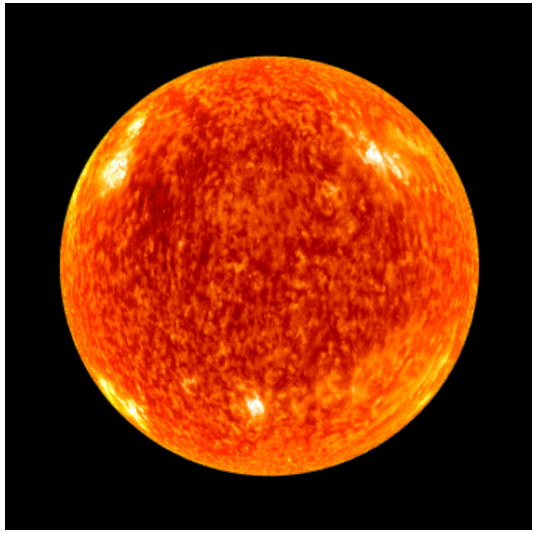
```
gl.depthMask(false);
```

- Then, to enable it again, you can use this code:

```
gl.depthMask(true);
```

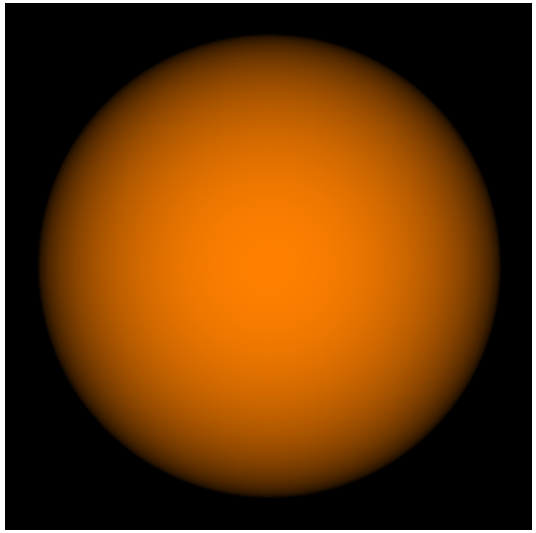
- You will disable the depth buffer, draw the outer glow geometry, before enabling the depth buffer and draw the Sun

# Combining the Sun and the Outer Glow



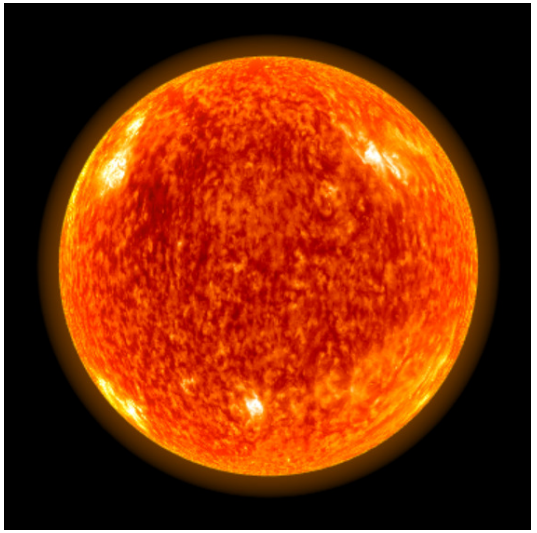
*The Sun with inner glow*

+



*The outer glow*

=



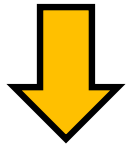
*The final result*

# Task 6. Rendering the Earth

- Right now the Earth is a white ball
- You can easily make it look like the Earth using texture mapping
- Two textures are given to you:
  - Colour map, the surface colour the Earth
  - Cloud map, the cloud distribution on the surface of the Earth

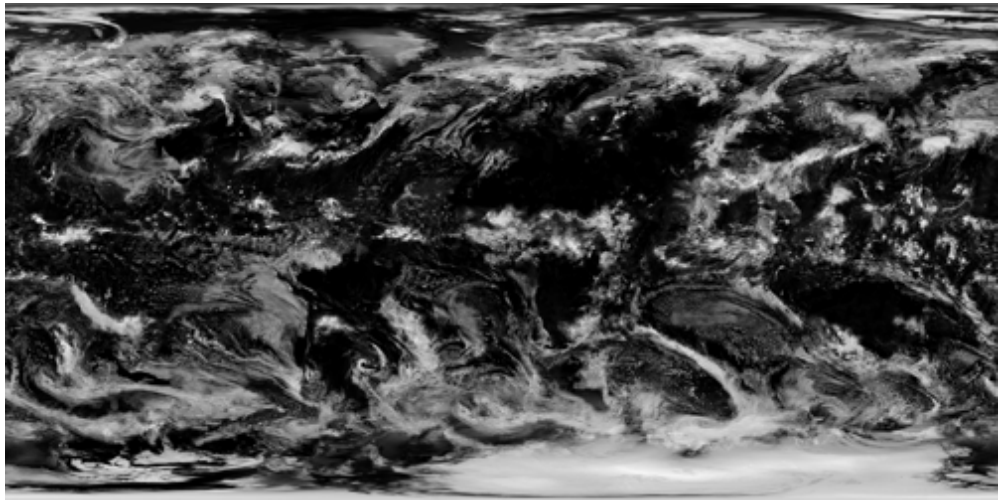
# The Earth Colour Map

- The Earth colour map is a simple texture
- Just as what you have done for the Sun, you can apply the texture easily
- This is the base colour of the Earth; a cloud layer will be added to make it more realistic



# The Earth Cloud Map

- The cloud map is an unusual map so that it contains only grayscale colour



- The colour values indicate the location and density of the clouds
- It is how much the cloud is covering the Earth as a percentage from 0 to 100%

# Using the Cloud Map

- Since the cloud map does not contain colours, you do not use it as a colour map
- You need to use the value of the map as an alpha value so that the colour shown on the Earth is:

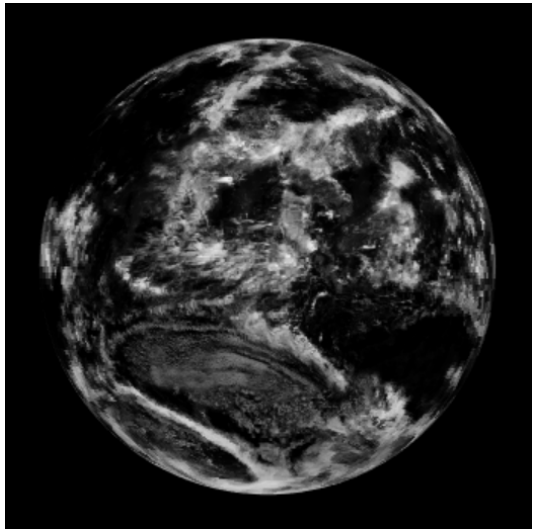
$$\text{texture colour} = \text{colour map} \times (1 - \text{alpha}) + \text{cloud colour} \times \text{alpha}$$

- The above equation gives you the final texture colour to combine with the lighting result
- You can assume the cloud is white in colour

# Mixing the Texture Maps

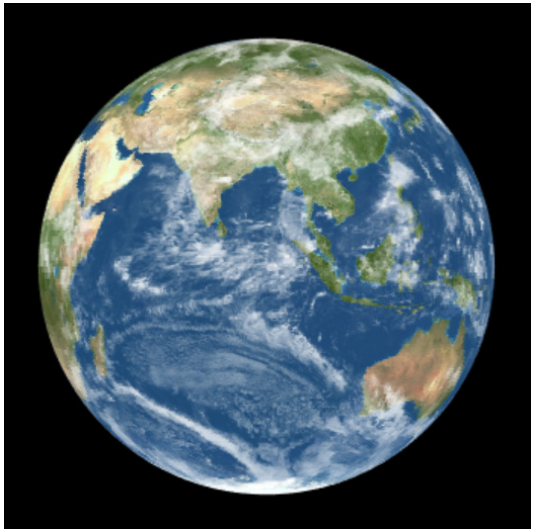
*The colour map*

+



*The cloud map*

=



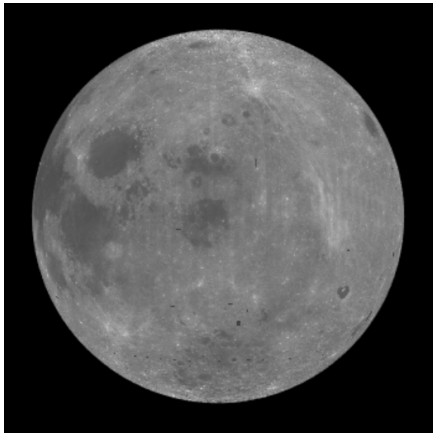
*The final result*

# Task 7. Rendering the Moon

- The Moon is a small white spot around the Earth
- You can make the Moon look realistic by using again two textures:
  - Colour map, the surface colour the Moon
  - Normal map, the normal perturbation values

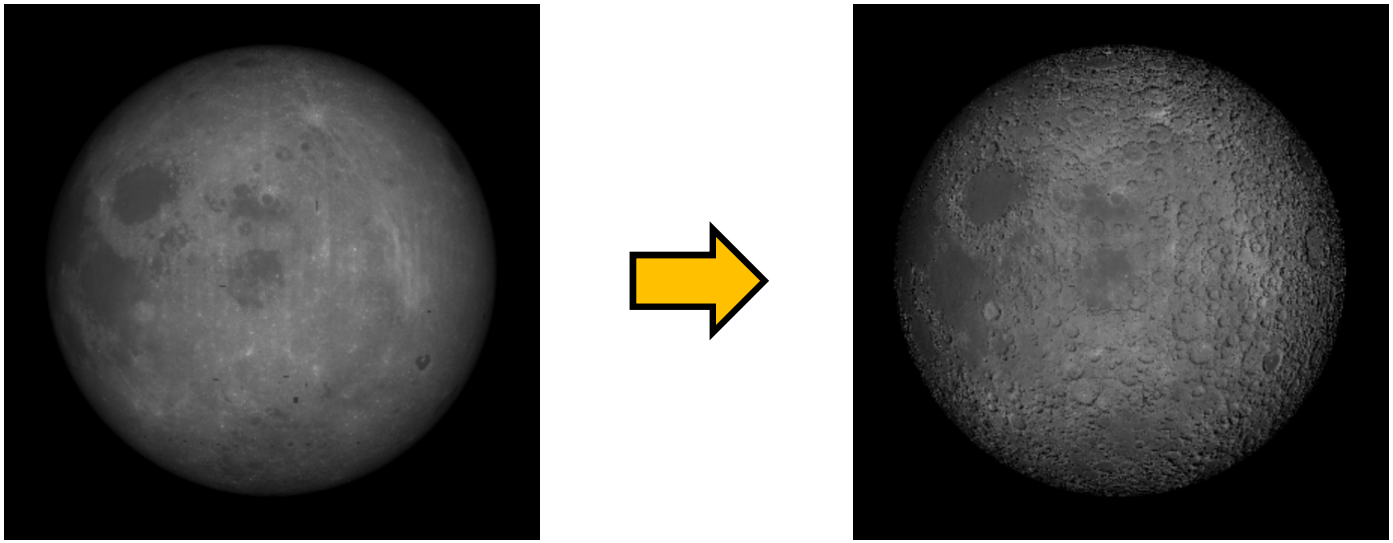
# The Moon Colour Map

- This is the third time you work on a simple colour map
- It should be very easy to do that now
- The colour map is again the base colour for the Moon so that you can modify it further



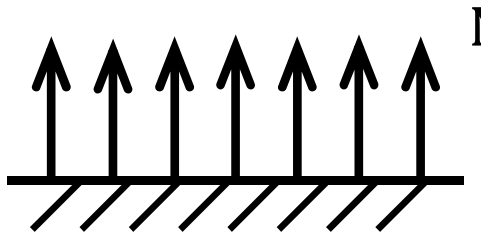
# Using a Normal Map

- It is good to show the colour of the Moon but it will be more impressive to represent the bumpiness of the Moon surface as well
- You will use a normal map to adjust the lighting calculation based on the surface roughness

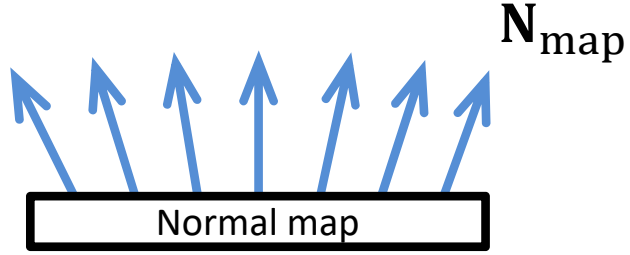


# Normal Mapping

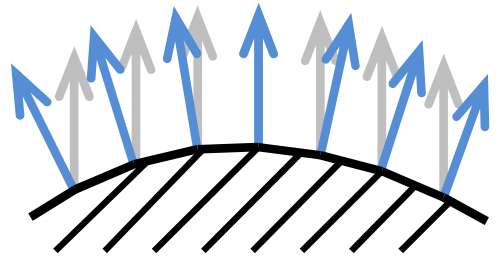
- In any given geometry, each vertex should have a normal which reflects the curvature of the actual surface
- Rather than those surface normals a different set of normals, from the normal map, are used for lighting calculation
- The perceived result is then a new and fake curvature created by the new normals




*Actual surface*



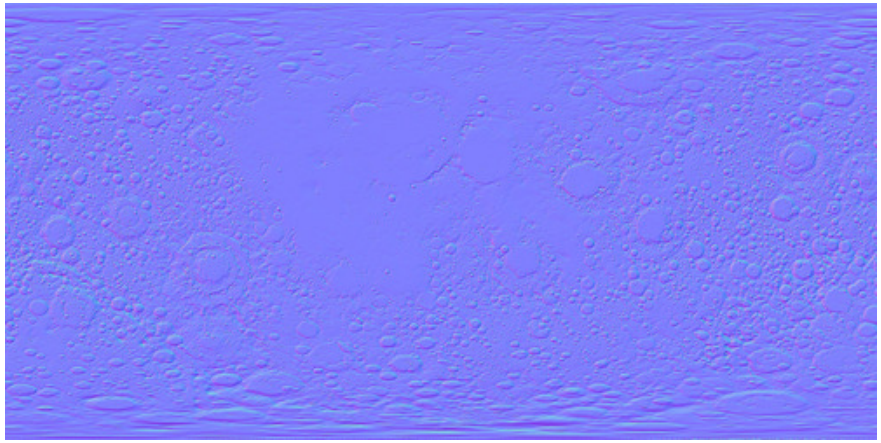

*Normals from the texture*



*Perceived surface as if it is curved like above*

# Content of the Normal Map

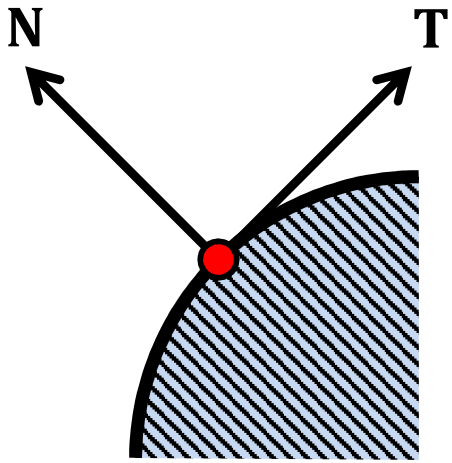
- You are given a normal map
- It does not look the Moon surface because it contains normals (not colours!)

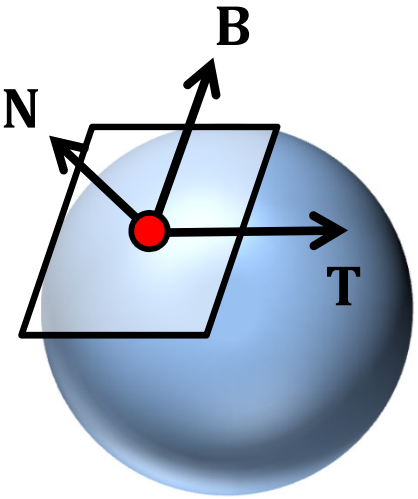


- Each pixel on the map is the normal at that particular position of the Moon
- The X, Y and Z components of the normal are cleverly stored in the R, G and B components in the texture
- The normal stored in the map is in the *tangent space* at that surface point

# The Tangent Space

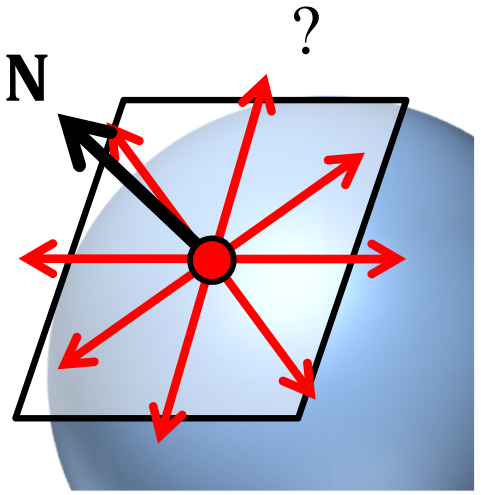
- For a surface, the normal $\mathbf{N}$ is a vector perpendicular to the tangent plane
- If we fix a vector $\mathbf{T}$ lying on the tangent plane, a tangent space can be created with three orthogonal vectors:
  - The surface normal: $\mathbf{N}$
  - The surface tangent: $\mathbf{T}$
  - The surface bitangent: $\mathbf{B} = \mathbf{N} \times \mathbf{T}$



# The Tangent Vector

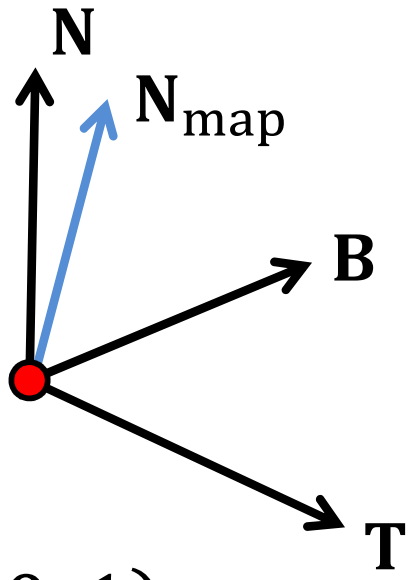
- On a surface point, there are infinite number of tangent vectors
- For a normal map, the tangent of the surface is typically the one that aligns with the **U** direction in the texture space




- The code to calculate the tangent for each vertex has been given in the starting code and the tangents are sent to the vertex shader as an attribute variable called `a_tangent`

# Reading the Colours of a Normal Map

- The colour of a normal map appears bluish because most of the time, the normals $\mathbf{N}_{\mathrm{map}}$ are pointing 'upwards' in the tangent space




- The coordinates of such normals in the TBN space are then very close to $(0, 0, 1)$
- This value is then adjusted to become $(0.5, 0.5, 1)$, i.e. the light blue colour, when it is stored
- If there are a lot of colours in the map, the surface is very rough; otherwise, the surface is relatively flat

# The Lighting Calculation

- In the shader that we have seen so far the lighting calculation has been carried out in the eye space
- However, the normal $\mathbf{N}_{\mathrm{map}}$ that you get from the normal map is in the tangent space
- There are two approaches – you either:
  1. Transform the normal $\mathbf{N}_{\mathrm{map}}$ to the eye space and then perform lighting calculation there
  2. Transform everything else to the tangent space and then perform lighting calculation there

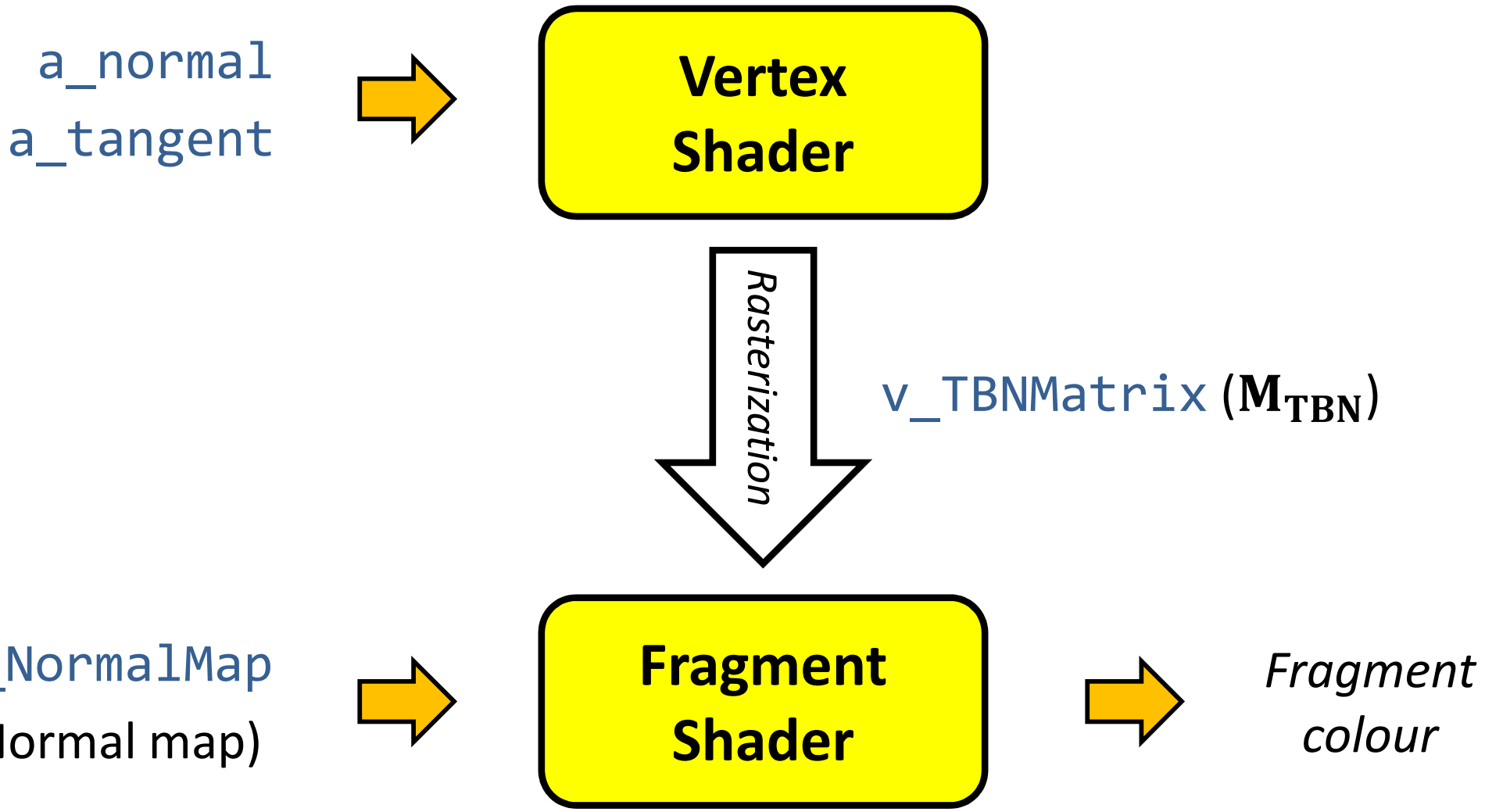
# Suggested Approach

- In the assignment, you will use the first approach, even though the second one is more efficient
- An advantage of using the first one is that it requires less changes to your code
- Either way, you will need to understand the tangent space and work out the $\mathbf{TBN}$ vectors in the vertex shader
- You will then use the vectors to transform your normals retrieved from the normal map

# Implementing Normal Mapping

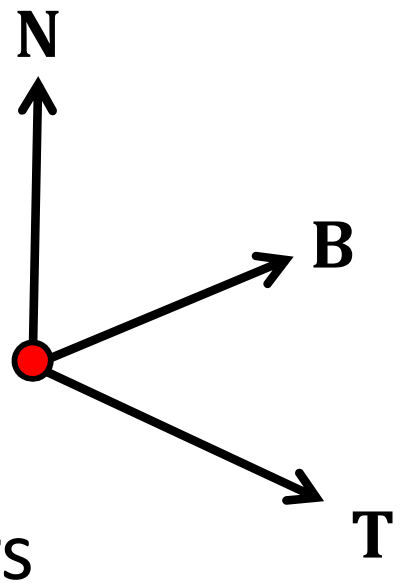
- Here are the steps:
  - In the vertex shader
    1. Finding the $\mathbf{T}$, $\mathbf{B}$ and $\mathbf{N}$ vectors
    2. Constructing the matrix $\mathbf{M_{TBN}}$ for transformation from the tangent space to the eye space
  - In the fragment shader
    3. Retrieving the normal $\mathbf{N}_{\mathrm{map}}$ from the normal map
    4. Transforming $\mathbf{N}_{\mathrm{map}}$ to the eye space using $\mathbf{M_{TBN}}$
    5. Calculating lighting using the transformed $\mathbf{N}_{\mathrm{map}}$

# Some Relevant Inputs and Outputs



# Finding the **T**, **B** and **N** Vectors

- Two of the **T**, **B** and **N** vectors are inputs of the vertex shader, i.e. they are attribute variables
- You need to be careful that some of the tangents (`a_tangent`) given to the vertex shader are ***NOT*** unit vectors




- You can find the remaining vector by computing the cross product of the other two
- The vectors form the tangent space at the vertex

# Transforming the $\mathbf{T}$, $\mathbf{B}$ and $\mathbf{N}$ Vectors

- The $\mathbf{T}$, $\mathbf{B}$ and $\mathbf{N}$ vectors obtained from the previous slide are expressed in the object space
- This is not useful because all other calculations are done in the eye space
- That means you need to transform them so that they are expressed in the eye space, i.e.:

$$\mathbf{T}_{\text{eye}} = \mathbf{M}_{\text{modelview}}\mathbf{T}$$
$$\mathbf{B}_{\text{eye}} = \mathbf{M}_{\text{modelview}}\mathbf{B}$$
$$\mathbf{N}_{\text{eye}} = \mathbf{M}_{\text{modelview}}\mathbf{N}$$

*Remember that they are vectors so you may not directly multiply the modelview matrix onto them*

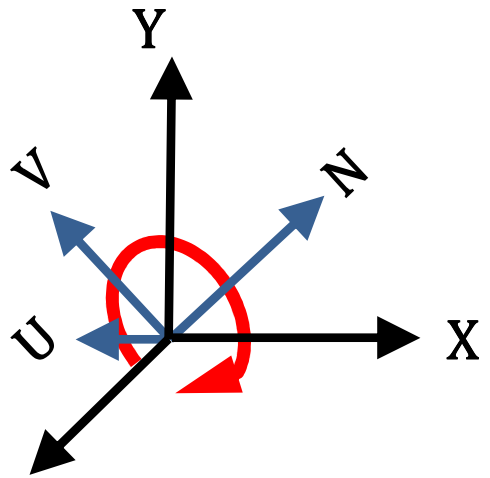
# Constructing $\mathbf{M}_{\mathbf{TBN}}$

- Once the TBN frame is ready, you can construct the matrix that is used to transform the normals $\mathbf{N}_{\mathrm{map}}$ to the eye space in the vertex shader
- Although the transformation is applied in the fragment shader it will be more efficient to do it once for each vertex in the vertex shader
- The matrix is then sent to the fragment shader as a varying variable `v_TBNMatrix`
- The following two slides that we have discussed before is useful for making $\mathbf{M}_{\mathbf{TBN}}$

# Change of Basis in 3D Space

*This is extracted from the "Transformations" notes*

- Given a set of basis $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{N}$, let's find a matrix to transform a point in the XYZ space to the UVN space
- Let $\mathbf{P_{XYZ}}$ be the point in the XYZ space and $\mathbf{P_{UVN}}$ be the point in the UVN space, then:




$$\mathbf{P_{XYZ}} = \mathbf{U}x_u + \mathbf{V}y_v + \mathbf{N}z_n = [\mathbf{U} \quad \mathbf{V} \quad \mathbf{N}] \begin{bmatrix} x_u \\ y_v \\ z_n \end{bmatrix} = \mathbf{MP_{UVN}}$$

$$\Rightarrow \mathbf{MP_{UVN}} = \mathbf{P_{XYZ}} \Rightarrow \mathbf{P_{UVN}} = \mathbf{M}^{-1}\mathbf{P_{XYZ}}$$

# Finding the Inverse of $\mathbf{M}$

*This is extracted from the "Transformations" notes*

- The matrix $\mathbf{M}$ is an orthogonal matrix, i.e. the column vectors are orthogonal to each other
- The inverse of such a matrix is then $\mathbf{M}^{-1} = \mathbf{M}^{\mathrm{T}}$
  - Let $\mathbf{U} = (u_x, u_y, u_z)$, $\mathbf{V} = (v_x, v_y, v_z)$ and $\mathbf{N} = (n_x, n_y, n_z)$ in the XYZ space, the matrix $\mathbf{M}^{-1}$ will look like this:

$$\mathbf{M}^{-1} = [\mathbf{U} \quad \mathbf{V} \quad \mathbf{N}]^{\mathrm{T}} = \begin{bmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{bmatrix}$$

- This will be useful later in the rendering pipeline

# Retrieving $\mathbf{N}_{\mathrm{map}}$ from the Normal Map

- You can use `texture2D` to get the normal $\mathbf{N}_{\mathrm{rgb}}$ from the normal map using the texture coordinates
- However, the components of the normal are usually adjusted so that they fall between 0 and 1
- To restore the normal values you simply multiply it by 2 and then subtract it by 1, i.e.:

$$\mathbf{N}_{\mathrm{map}} = \left(2 * \mathbf{N}_{\mathrm{rgb}}\right) - 1$$

- It is then transformed by $\mathbf{M}_{\mathbf{TBN}}$ and lighting is computed using the new normal

# Writing GLSL Code 1/2

- We have discussed some data types and the swizzling operation in GLSL
  - It is likely you will need to use them in your assignment
- GLSL is strict about data types
- That means you will need to be careful when using operators
  - For example, if you have a float variable `x`, writing the condition `x > 0` will result in an error
  - The correct code should be `x > 0.0`

# Writing GLSL Code 2/2

- The GLSL functions that you will use in the assignment have all been used in the course examples before, such as `dot()`, `normalize()`
- You have to be careful when specifying a matrix, for example, when you work with the TBN matrix
- This is because GLSL uses a column-major order when storing matrices
- Finally, debugging is not easy in GLSL because you do not have any output console to work with

# Debugging GLSL Code

- You cannot output debug messages in GLSL
- A quick way or the only way to debug is then to show the intermediate results visually
- For example, to inspect the input tangent values to the vertex shader you can:
  - Pass a debugging varying variable, for example `v_Tangent`, to the fragment shader
  - Put the varying variable as the fragment colour
- An example is shown in the following slide

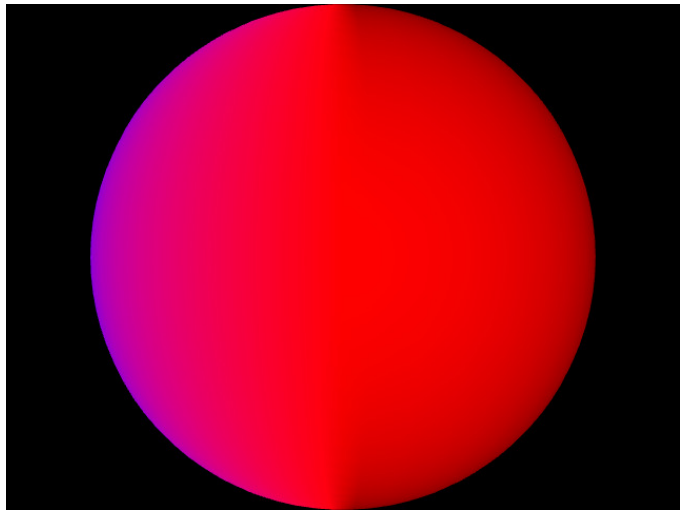
# Showing the Tangent Values

- Vertex shader

```
attribute vec3 a_position;
attribute vec3 a_tangent;
uniform mat4 u_ModelViewProjMatrix;
varying vec3 v_Tangent;

void main() {
    v_Tangent = a_tangent;

    gl_Position =
        u_ModelViewProjMatrix *
        vec4(a_position, 1);
}
```

- Fragment shader

```
precision mediump float;
varying vec3 v_Tangent;

void main() {
    gl_FragColor =
        vec4(v_Tangent, 1);
}
```



# Marking Scheme Overview 1/4

- Hierarchical structure
  - The Sun, the Earth and the Moon are correctly created at their initial positions and sizes 12%
- System animation control
  - Pressing the spacebar key starts/stops the animation 2%
  - Pressing the '+' key or '-' key increases/decreases the speed of the animation 4%

# Marking Scheme Overview 2/4

- System interactions
  - The Sun rotates correctly 4%
  - The Earth rotates correctly 4%
  - The Earth orbits correctly 4%
  - The Moon orbits correctly 4%
  - The Moon faces the Earth from the same side 2%
- Camera positioning
  - Pressing ‘1’ puts the camera in front of the Sun 2%
  - Pressing ‘2’ puts the camera in front of the Earth 4%
  - Pressing ‘3’ puts the camera in front of the Moon 4%

# Marking Scheme Overview 3/4

- Using the Sun as the light source
  - The Sun is always an emitter 2%
  - The Earth is shaded correctly 4%
  - The Moon is shaded correctly 4%
- General texture mapping
  - A basic colour map is applied correctly for each of the Sun, the Earth and the Moon 12%

# Marking Scheme Overview 4/4

- The Sun rendering
  - An inner glow is created inside the silhouette 4%
  - An outer glow is created outside the silhouette 8%
- The Earth rendering
  - A cloud map is shown appropriately 8%
- The Moon rendering
  - Surface roughness is shown on the surface 8%
  - Lighting is calculated correctly when using the normal map 4%

# Submission

- The deadline of the assignment is:

**8pm, Thursday, 9 May 2019**

- To submit your assignment:
  - You need to put **everything** (HTML file, JavaScript files and any other files) into a zip file called `<your ITSC account>_a2.zip`
  - For example, if your ITSC account is *johnc*, you will put your files into `johnc_a2.zip`
  - You can then submit the zip file through canvas