

Computer Graphics (MIEIC)

2019/2020

Final Project

v2.2 (2020/05/06)

Objectives

- Apply the knowledge and techniques acquired to date
- Create a complex scene, with different geometries, materials, textures and lights
- Explore use of shaders, interaction, keyboard control and animation

Description

This project aims to create a scene that combines the different elements explored in previous classes, adding some new elements. For this work you must start with the base code provided in Moodle, which corresponds to an empty scene. You will need to add some of the objects created earlier later.

At the end, the scene will generally consist of:

- A surrounding landscape (*cube map*)
- A vehicle controlled by the user
- A terrain
- Other elements

The following points describe the main characteristics of the different required elements. Some freedom is given regarding their composition in the scene, so that each group can create its own scene.

The worksheet is divided into two parts, with a proposal for a work plan at the end of each one.

Part A

1. Creating base objects

1.1. MyCylinder - Cylinder (without tops)

Create a new class **MyCylinder** that approximates a cylinder with a radius of one unit, a variable number of "sides" (**slices**), and height of one unit (in Y). The cylinder base must coincide with the XZ plane and be centered on the origin.

The normals of each vertex must be defined in order to approximate a curved surface, according to Gouraud's Smooth Shading method, as shown in **Figure 1**.

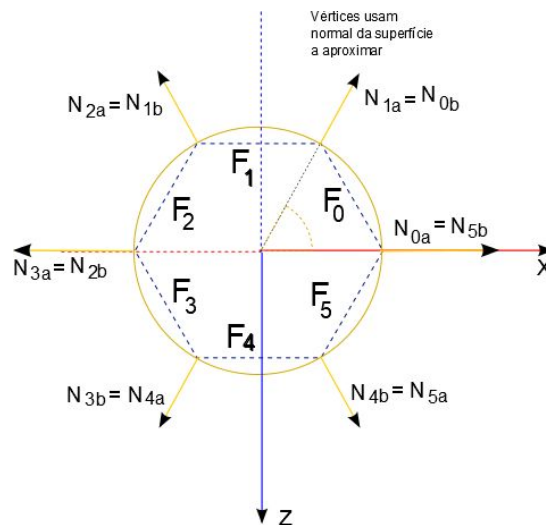


Figure 1: Illustration of the normals to be assigned to each vertex in the case of a cylinder approached from six sides.

For each vertex, define the texture coordinates to map a texture around the cylinder. Note that as the texture goes around the cylinder you may need to repeat the first vertical line of vertices as they are, simultaneously, the beginning and the end of the texture.

1.2. MySphere - Sphere

In the code provided you will find a MySphere class corresponding to a sphere with the center at the origin, with a central axis coinciding with the Y axis and unit radius.

The sphere has a variable number of "sides" around the Y axis (slices), and "stacks" (stacks), which correspond to the number of divisions along the Y axis, from the center to the "poles" (i.e., number of "slices" of each semi-sphere). **Figure 2** is a visual representation of the sphere.

It is intended that you analyze the code of this object, and add the code necessary to generate the texture coordinates to apply textures to the surface of the sphere, as shown in **Figure 3**.

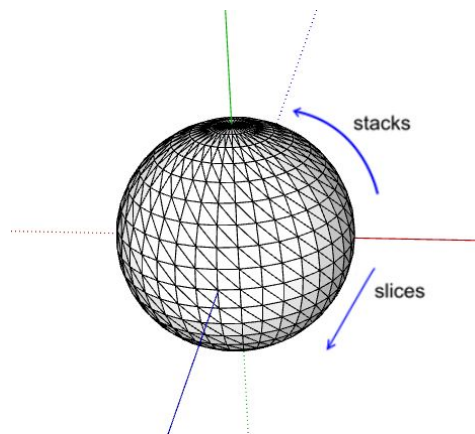


Figure 2: Example image of a sphere centered on the origin.

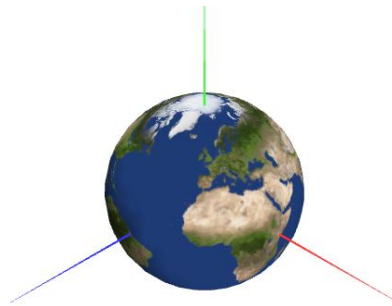


Figure 3: Example of texture (available in the base code) and its application in asphere

1.3. Cube Map - Creating a cubemap for the environment

In this exercise, a *cube map* will be created, which will serve as the background for the scene. A *cube map* can be defined as:

- a large cube (much larger than that of the visible scene),
- with zero specular and diffuse component, and strong ambient component,
- with only the inner faces visible,
- and its mapped textures represent the surrounding environment of a scene (e.g., a landscape; see **Figure 4**).

This object can be obtained in two different ways (you must choose one):

- using **MyUnitCube** class as base, with a single texture similar to Figure 4, and mapping different parts of the texture to each face of the cube or
- using **MyUnitCubeQuad** class as base, and six different textures, one for each face of the cube

Choose one of these two options and include in the project folder a copy of the respective class code - **MyUnitCube** or **MyUnitCubeQuad**. Modify the copy to create a new class **MyCubeMap**, in order to be visible from the inside, and with texture coordinates according to this option.

The *cube map* must be unitary and when used in the scene, it must be scaled to measure **50 units on its side**. If necessary, you may need to change the position of the camera so that it is inside and centered inside the *cube map*.

The base code includes an example of images for each of the two types of *cube map* (one similar to Figure 4, and a pack of 6 images for the option of six *quads*).

You should look for/create at least one more image surrounding your choice, to then allow the user to choose between that image and the example image, through the graphical interface (see point 2.2). As a starting point, you can check the following web address:

<https://www.cleanpng.com/free/skybox.html>

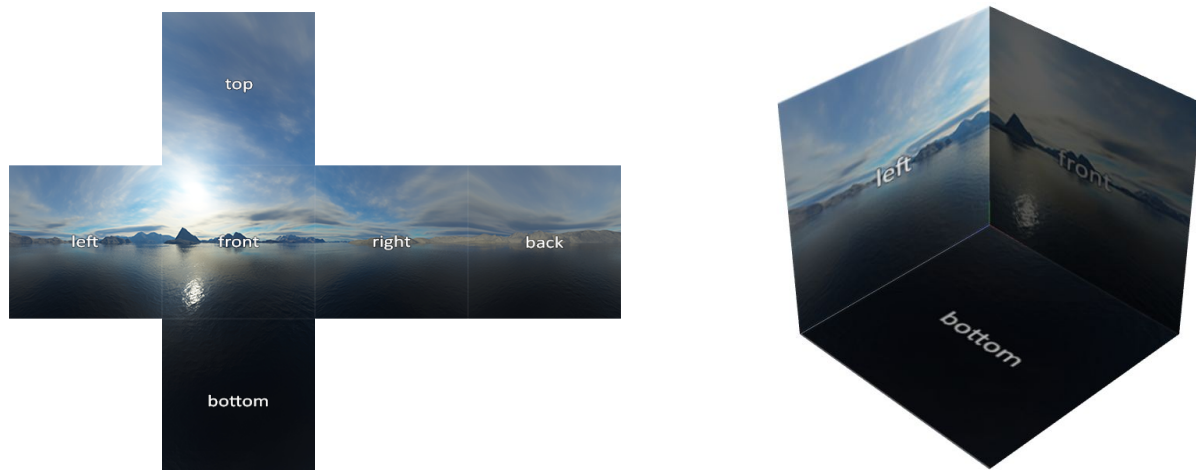


Figure 4: Image of the skybox type, and its application in a cubemap viewed from outside

1.4. *MyVehicle* - Preliminary version

Create the class **MyVehicle**, which will represent the vehicle in the scene. The final constitution of the vehicle will be detailed in Part B of the worksheet. At this stage, only a basic representation should be created to identify its position and orientation. For this purpose, we suggest that you use a simple shape, such as a non-equilateral triangle like the **MyTriangle** created for the Tangram, or the **MyPyramid** provided in TP3 (see **Figure 5**).

Regardless of the chosen geometry, the following must be guaranteed:

- Centered at the origin,
- Have unit size
- Have the central axis aligned with the positive axis of the ZZ, that is: have the front pointing in the direction of Z+, as shown in **Figure 5**.

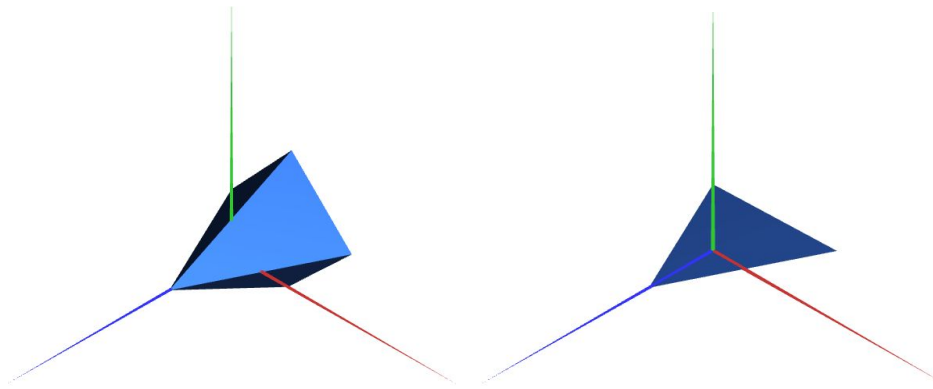


Figure 5: Exemplification of the draft version of MyVehicle
a) Pyramid b) Triangle in the XZ plane

2. Interface

2.1. Vehicle control using keyboard

In order to be able to control the movement of the vehicle in the scene using key presses, it will be necessary to change:

- The interface, to detect the pressed keys
- The scene, to invoke control functions on the vehicle according to the detected keys
- The vehicle, to include control functions such as “rotate” and “move” that change the position and orientation of it.

The steps necessary for this purpose are presented in more detail below.

IMPORTANT NOTE: If you use *copy-paste* with the following code, some symbols may not be well recognized (although they look the same) and cause Javascript errors. Avoid doing so, or check the code well before proceeding.

1. Change the class **MyInterface**, adding the following methods to process multiple keys at the same time:

```
initKeys() {
  // create reference from the scene to the GUI
  this.scene.gui=this;

  // disable the processKeyboard function
  this.processKeyboard=function(){};

  // create a named array to store which keys are being pressed
  this.activeKeys={};
}
```

```

processKeyDown(event) {
    // called when a key is pressed down
    // mark it as active in the array
    this.activeKeys[event.code]=true;
};

processKeyUp(event) {
    // called when a key is released, mark it as inactive in the array
    this.activeKeys[event.code]=false;
};

isKeyPressed(keyCode) {
    // returns true if a key is marked as pressed, false otherwise
    return this.activeKeys[keyCode] || false;
}

```

NOTE: At the end of *init()* function in **MyInterface**, call the *initKeys()* function.

2. In the **MyScene** class, add the following *checkKeys()* function and add a call to it in the *update()* function.

```

checkKeys() {
    var text="Keys pressed: ";
    var keysPressed=false;

    // Check for key codes e.g. in https://keycode.info/
    if (this.gui.isKeyPressed("KeyW")) {
        text+=" W ";
        keysPressed=true;
    }

    if (this.gui.isKeyPressed("KeyS")) {
        text+=" S ";
        keysPressed=true;
    }
    if (keysPressed)
        console.log(text);
}

```

Run the code and check the messages on the console when “W” and “S” are pressed simultaneously.

3. Prepare the class **MyVehicle** to move the vehicle:
 - Add variables in the constructor that define:
 - the vehicle's orientation in the horizontal plane (angle around the YY axis)
 - its speed (initially at zero)
 - its position (x, y, z)
 - Create the **MyVehicle.update()** function to update the *position* variable according to the *orientation* and *speed* values.
 - Use the *position* and *orientation* variables in the **display()** function to orient and position the vehicle.
 - Create the functions **turn(val)** and **accelerate(val)** to change the orientation angle, and to increase/decrease the speed value (where **val** can be a positive or negative value).
 - Create the function **reset()** that should return the vehicle to its initial position, with zero speed and rotation.
4. Change the **checkKeys()** function in **MyScene** to invoke the functions **turn()**, **accelerate()** or **reset()** of the vehicle to implement the following behavior depending on the keys mentioned:
 - **Increase or decrease the speed** according to pressing “W” or “S”, respectively.
 - **Rotate the vehicle** to the left or right by pressing the keys “A” or “D” respectively.
 - Pressing the “R” key should invoke the **reset()** function of the vehicle.

2.2. Additional controls on the interface

Add the following additional controls on the graphical interface (GUI), to be able to parameterize the movement of the vehicle and the appearance of the *cube map*:

1. Create a *dropdown list box* containing the **different environment textures**, so that the user is able to change the texture of the *cube map* at run time (suggestion: follow the example of textures in TP4).
2. Create a *slider* in the GUI called **speedFactor** (between 0.1 and 3) that multiplies the vehicle's travel speed, each time the "W" or "S" keys are pressed.
3. Create another *slider* in the GUI called **scaleFactor** (between 0.5 and 3) that allows you to scale the vehicle so that it is easier to observe its animations.

Proposed work plan - Part A

- Week of March 30: Beginning of point 1
- Week of April 6: Continuation of point 1; Point 2.1
- Week of April 13: Point 2.2 (+ beginning of Part B, to be published by then)

Part B

3. *MyVehicle - Blimp*

3.1. *Modeling the Blimp*

Change the class **MyVehicle** to represent a blimp. To model the blimp, you can use a combination of the different objects created previously, so that the blimp consists of a body, gondola (passenger compartment) with two propeller engines, and rudders (directional “fins” at the back of the blimp). **Figure 5** illustrates the type of structure with simple geometries.

The blimp to be developed can use geometries different from the example, however it should not have an excessive number of polygons (for performance reasons). We suggest using cylinders and spheres, but new objects may be created that are applicable to the blimp’s model.

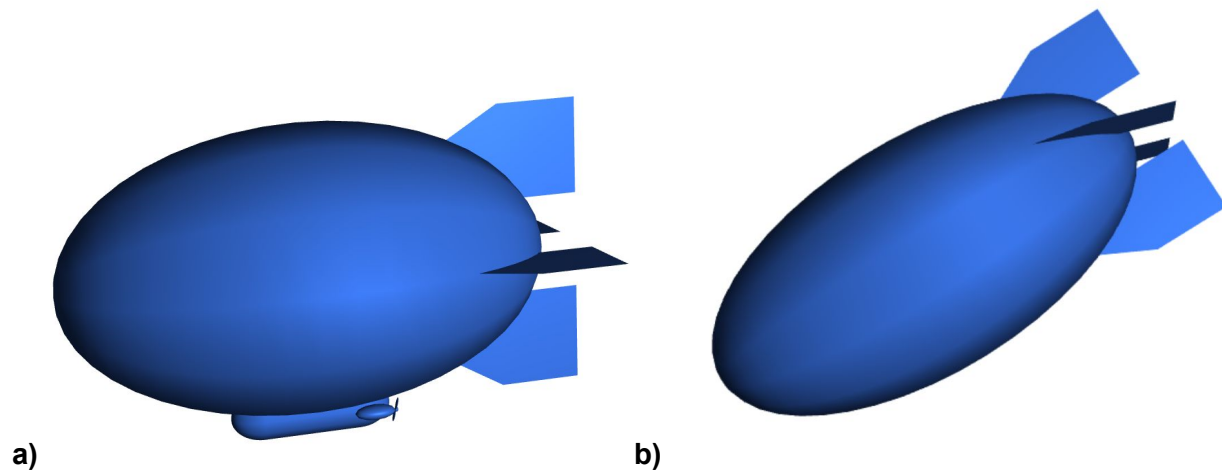


Figure 5: a) Modeling of the blimp, created with spheres, cylinders, triangles and squares. b) top view with inclined rudders (see point 3.2)

The blimp must be visible when the scene starts, in order to facilitate its evaluation, as well as the other elements of the scene. Place it so that the center of the ellipsoid is in the (0, 10, 0) position. The main body should be approximately 2 units long and 1 unit wide and high. The different objects used to create the blimp must have materials and textures applied to them, appropriate to the parts of the blimp they represent.

Submit an image of the blimp's appearance that displays its details and textures.



(1)

3.2. Additional Blimp animations

The goal is to create new animation and control mechanisms for the class **MyVehicle**, in addition to the functionalities created in Part A of the worksheet.

1. The **propellers** of the blimp should have a rotation animation, ideally proportional to the current blimp's speed.
2. The **vertical rudders** should tilt towards the left or right when the blimp is changing its direction (see **Figure 5.b**). While the "A" or "D" key is pressed, the rudders should tilt in the opposite direction of rotation.
3. **Autopilot**: Create a new blimp animation method, which is activated / deactivated when the "P" key is pressed. While active, the other keys will be ignored, with the exception of the *reset* key ("R").

This method represents an autopilot mode, during which the blimp moves in consecutive circles with **5-unit radius**, **starting from the current position when autopilot starts**.

To implement this animation, shown in **Figure 6**, it is necessary to:

- Determine the center of the animation circle from the current position and orientation;
- Determine initial angle in relation to the direction parallel to the XX axis;
- Define the starting position using the calculated center and initial angle;
- Update the position and orientation for each frame so that the blimp moves in a circle, with orientation equal to the tangent of the circle at the current position;
- **A complete lap should take 5 seconds, regardless of the performance of the computer used.**

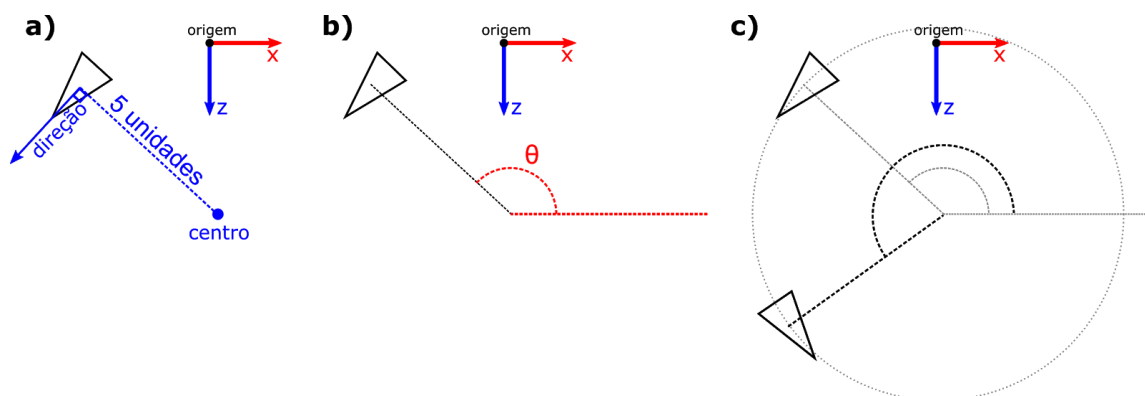


Figure 6: Exemplification of the steps of the animation "autopilot"

4. Terrain

In this exercise, a terrain will be added to the scene, created with a simplified version of the water shaders created in TP 5.

The terrain will be built using two textures:

- One that works as a heightmap (similar to the wave map used in the water)
- Another one with the colors to be mapped on the terrain

Figure 7 contains a possible example of **MyTerrain**, with the respective textures.

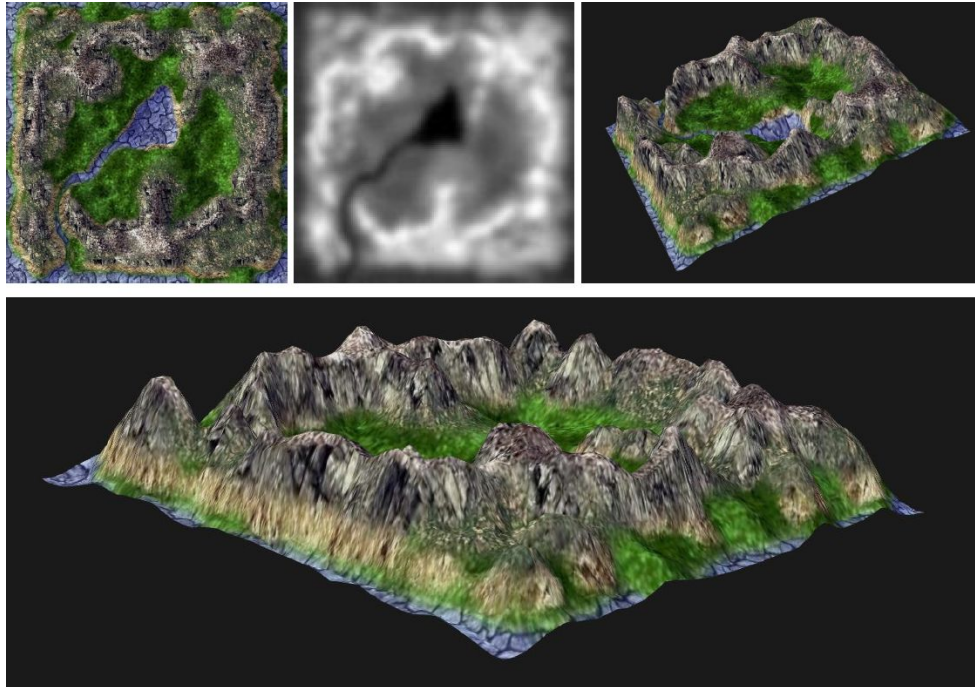


Figure 7: Color texture, height map and generated geometry.
(source: Outside of Society http://oos.moxiecode.com/js_webgl/terrain/index.html)


To achieve this effect, some suggestions are to:

- Create copies of the “water.frag” and “water.vert” shader files, With the new names “terrain.frag ”and “terrain.vert”;
- Remove the code parts related to the animation from the new shaders;
- Create a new **MyTerrain** class, which will consist of a **MyPlane** with 20 divisions in each direction (see shaders’ practice class). The terrain class contains, initializes and uses the “terrain.frag” and “terrain.vert” shaders prepared in the previous points;
- Create an instance of MyTerrain in the scene with dimensions of 50x50 units, and with a maximum height of 8 units.
- Test an initial version with the color and heightmap textures provided.
- Modify the heightmap texture (using an image editor) so that there is a central flat area of approximate dimensions 20x20 units (height at zero, Y = 0). Alternatively, you can

replace the provided textures with another pair of textures of color and height for the same purpose - maintaining the central flat area as described.

In the class **MyScene**, change the code so that:

- If necessary, apply Y translation on the terrain or on the cube, to align the cubemap images with the terrain;
- The initial position and orientation of the camera allows you to see the blimp and most of the flat area of the terrain.

Capture a perspective of the scene containing the background created with the cubemap, the terrain, and vehicle.  (2)

5. *Launching of supplies*

A new functionality will be added to **MyVehicle**, so that the blimp can drop supplies (new **MySupply** class).

5.1. Creation of **MySupply** class

Create a new **MySupply** class, which will be based on the code of the **MyUnitCubeQuad** class (that is, it will have the same code initially, which must be changed as indicated below), with materials and textures to be defined by each group (e.g., appearance of wooden box). This class should include the following states:

- INACTIVE: When not yet launched - it is not displayed in the scene;
- FALLING: When it is launched and while it is falling - it is displayed with the material and textures mentioned above, and animated as described in point 5.3;
- LANDED: When it falls on the ground, it is displayed differently to represent the result of the fall (e.g., with its six faces "spread out" on the floor / opened box).

It is suggested that you implement a simple state machine within the class:

- Define an enumeration of possible states, and in the constructor initialize a variable that will represent the state.

```
const SupplyStates = {
  INACTIVE: 0,
  FALLING: 1,
  LANDED: 2
};
constructor ()
{
  this.state=SupplyStates.INACTIVE;
}
```

- In the *MySupply.update()* and *MySupply.display()* functions, use the value of *this.state* to choose the type of changes/appearance to be applied (for example, the *update()* function will only need to update the box position when it is falling, in other states you probably won't have to do anything).
- Implement the necessary methods to trigger state changes:
 - *drop(dropPosition)* - to change from INACTIVE to FALLING and start the drop animation from the *dropPosition* (current blimp position)
 - *land()* - to be called when the drop animation ends, to determine whether it has reached the XZ plane ($Y = 0$) or not. If this is the case, you should switch from FALLING to LANDED.
 - Other methods that you consider relevant (for example, and as a suggestion only, auxiliary design functions may be used - *displayFalling*, *displayOnLanded* - which are called by the *display()* function, depending on the state).

5.2. Launch Control

Add to **MyScene** the functionality of launching 5 supplies from the blimp, which should work as follows:

- When pressing the “L” key, a **MySupply** object should appear at the bottom of the blimp, and start its fall;
- The supply should take 3 seconds to reach the XZ (Y null) plane;
- When the supply reaches the XZ plane, it must have a different appearance/design (see LANDED state in point 5.1)

Note: the reset function (“R” key) must also reset supplies, removing them from the scene and making them available for new launches.

It is suggested for this purpose that you change the following in the **MyScene** class (non-exhaustive suggestions, you may have to add / change other features):

- Add a list of 5 **MySupply** objects, all of which are inactive initially;
- In the *update()* and *display()* functions of the scene, add the invocation of the respective *update()* and *display()* methods for each of the 5 objects;
- Add the necessary code in the *checkKeys()* function to detect the “L” key and:
 - Invoke the *drop()* method of an inactive object, providing the current position of the blimp, thus initiating its fall;
 - Update an internal counter variable *nSuppliesDelivered* that indicates the number of supplies that have already been launched (to be used in point 6).

Capture an image of the scene where a falling supply is visible (with the state *FALLING*), and another in which a supply already on the ground is visible (with the state *LANDED*). Use the camera zoom to ensure the details of the supply objects are distinguishable.



(3)



(4)

6. Shaders

6.1. Flag on the blimp

Add an object from the class **MyPlane** to the class **MyVehicle**, and place it behind the blimp body, similar to the example in **Figure 8**. Change the **MyPlane** class so that the plane is double-sided..

Create and initialize shaders in **MyVehicle** that will be applied to the new flag object, which should:

- Change the height of the plane's vertices in order to recreate a “ripple” effect on the object, following a sinusoidal function (Suggestion: use the texture coordinates as parameter);
- The ripple effect should be animated, and the speed of the animation should depend on the speed of the blimp;
- Apply a texture of your choice with the *fragment shader*.

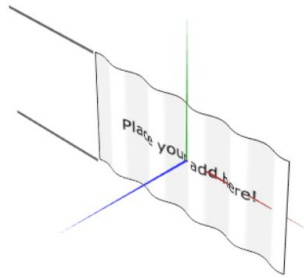


Figure 8: Example of a flag with a ripple and a simple applied texture

Capture an image that demonstrates the blimp with the created flag, in which it is possible to observe the ripple in it.



(5)

6.2. Supplies delivered counter

Create a new class **MyBillboard**, to represent a billboard with the number of launched supplies (**MySupply**), as counted by the variable *nSuppliesDelivered*.

This billboard should consist of (see **Figure 9**, with illustrative dimensions):

- a base plane with a decorative texture that includes the text “Supplies Delivered”;
- A progress bar (details below);
- two beams that support the geometry above.

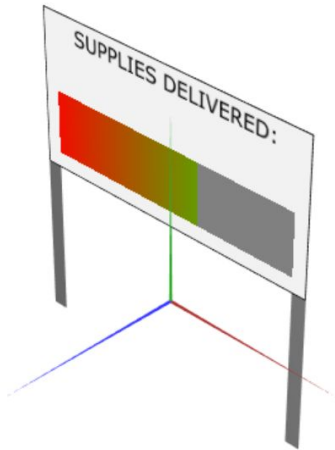




Figure 9: Example of a billboard with a progress bar

The base plan must have dimensions 2 x 1 units, the beams must be one unit high, and the entire billboard must be placed on the terrain in a position that is clearly visible from the camera's initial point of view.

The progress bar must consist of a single plane of dimensions 1.5 x 0.2 units (width x height) associated with a pair of shaders developed specifically for this purpose and activated in the class **MyBillboard**, following these instructions (see **Figure 10**):

- When the progress bar is completely filled (that is, all supplies delivered), it must display a horizontal gradient that varies from red (RGB = 1.0.0) to green (RGB = 0.1.0). This gradient must be calculated in the *fragment shader*;
- When only a part of the supplies have been delivered, only the corresponding percentage of the gradient should be visible. For example, four supplies delivered in a total of five will correspond to 80% of the total gradient;
- The rest of the area in the progress bar should be filled with a constant color of your choice, as long as it contrasts with the gradient tones (in the example, RGB gray = 0.5,0.5,0.5).

Capture an image of the billboard when three supplies have been launched.  (6)

Submit the final code.  (1)

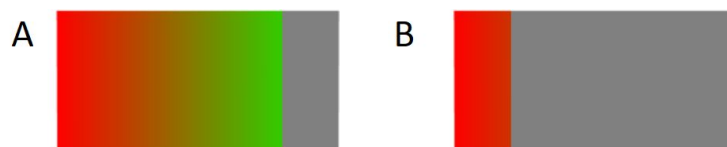


Figure 10: Examples of progress bar with gradient from red to green, drawn to 80% (image A) or 20% of the plan (image B)

Notes on the evaluation of the work

The maximum classification to be attributed to each item corresponds to an optimal development of it, in absolute compliance with all the features listed. Without loss of the desired creativity in a work of this type, any developments other than those requested will not be accounted for (as a way to compensate for other missing components), during evaluation.

1. Creation of base objects (3 points)

- 1.1. MyCylinder (1.5 points)
- 1.2. MySphere (0.5 points)
- 1.3. Cubemap (1 point)
- 1.4. MyVehicle (*0 points, counted in point 3*)

2. Interface (3 points)

- 2.1. Vehicle control using keyboard (2 points)
- 2.2. Additional controls on the interface (1 point)

3. Blimp (4 points)

- 3.1. Blimp Modeling (2 points)
- 3.2. Additional animations (2 points)

4. Terrain (1 point)

5. Launching supplies(3.5 points)

- 5.1. Creation of MySupply (2 points)
- 5.2. Launch Control (1.5 points)



6. Shaders (3.5 points)

- 6.1. Flag (2 points)
- 6.2. Supplies delivered counter (1.5 points)

7. Software structure and quality (2 values)

Checklist

Until the end of the work you should save and later submit the following images and versions of the code via Moodle, **strictly respecting the naming rule**:

-  **Images (6): (named as "proj-t<class>g<group>-n.png")**
-  **Code in zip file (1): (named as "proj-t<class>g<group>-n.zip")**