Interactive Graphics
(6 cfu)

Title: “Snake Project”

Students
Lorenzo Sciarra
Angelo Marvulli
Angela Longo

Professor
Marco Schaerf

SNAKE Project

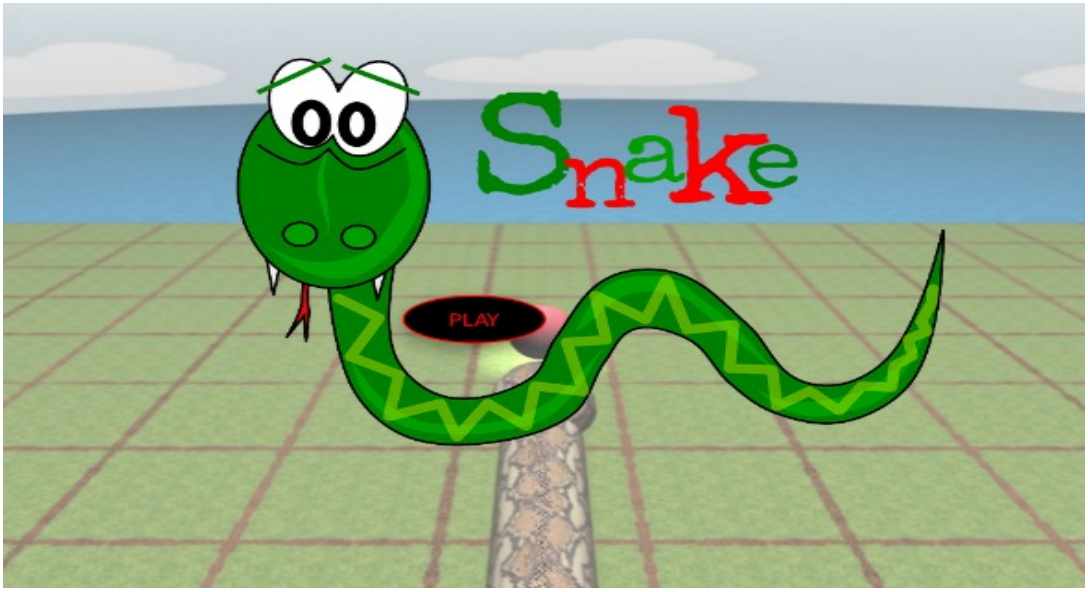
General description

Our project consists in the simulation of the game "SNAKE" , where a snake eats apples and grows until he goes out of the grid or it eats itself. The user interacts with the game, he moves the snake and he can lose if the conditions mentioned above occur or he can win if the snake eats a fixed number of fruits: in both cases the user may choose to restart the game. There is also a minimap that helps the user locating the position of the snake and of the apple in the map since the camera follow the view of the snake and does not give a full vision of the world.

Environment

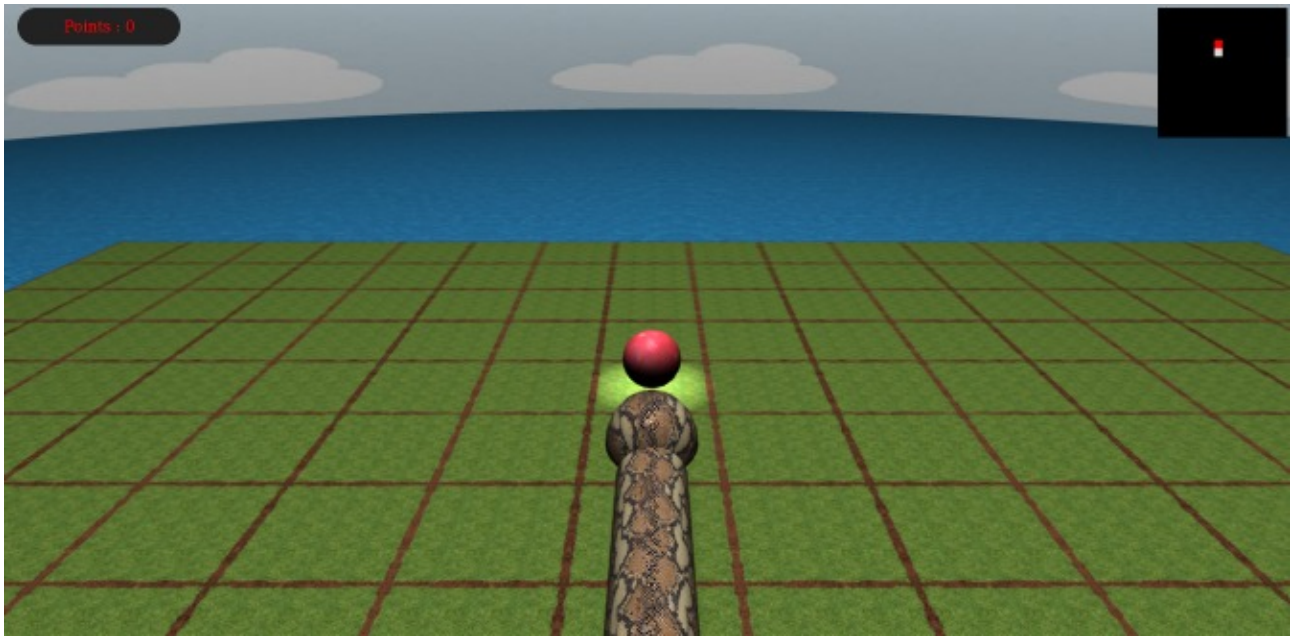
In the project we have used only basic WebGL.
The application consists in HTML files, which define the layout of the various pages, and JavaScript files, which define the logics behind the game.

At the beginning there is an html page *"start.html"*, which contains a button PLAY that, if pressed, starts the game calling the page "*main.html"*, which contains all the elements of the game.



The initial function is the window.onload() function in which the WebGL canvas is loaded and it is adapted to the window of the browser page and it initializes all the element of the environement: initShaders is used to load,

compile and link shaders to form a program object. We define shaders in html and then we compute and specify data in the js files, the data are sent to GPU and then with the function render, which calls itself recursively, they are rendered.



The **objects**, as we can see in the figure, are the world, the sea, the sky, the apple and the snake. Objects vertices, normals and texture coordinates lists are optimized by the use of index arrays and triangle strips logic, which allow to save a lot of memory at every rendering cycle.

The **camera** is placed behind snake head and it rotates and translates simultaneusly with its rotation and translation. We use the projection matrix to define the view volume and the camera lens.

The **lights** influence color and material properties of the objects. We have choosen to use both Phong and Gouraud Models. The first is used to render objects which are near to the camera, while the second is used for far ones. This also improves performance, because Phong model is heavier computationally, and far objects don’t show much difference if they are enlighted using Gouraud model.

The **textures** are loaded using images and every object type has its own.

The snake is organized in a **hierarchical model**: it is composed of a head, several body parts, and a tail.

Technical aspects

Every object type has its own vertices, normals, texture coordinates and indices arrays, which are passed to the shaders through array attributes. Additionally, every object has a model matrix and its normal, which are used to calculate the right position and illumination of the object vertices in the environment.

World

This object represents the platform on wich the snake moves.
The world appears like a square divided in a grid of 16x16 and its cells represent the positions through which the snake can pass and in which apples can appear.

Bonus

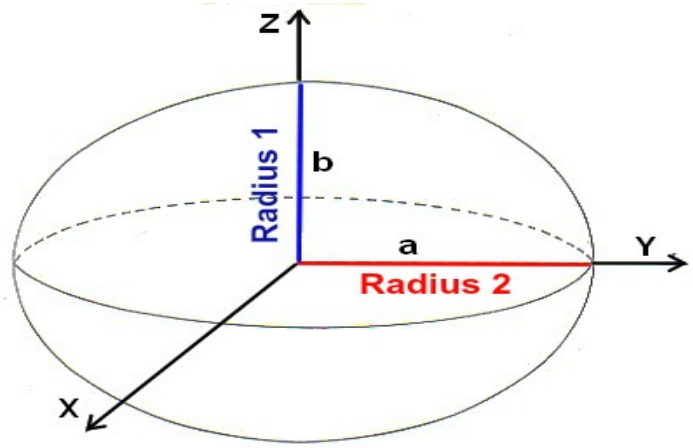
This object represents apples, and it is built using spherical coordinates:

$x = r\ \sin\theta\ \cos\varphi$, $y = r\ \cos\theta$, $z = r\ \sin\theta\ \sin\varphi$

Apples also have an animation which makes them rotate around their y axis.

Snake

This object is organized in different pieces.
The **head** is built using ellipsoidal coordinates, which are similar to polar ones but they involve the two axes of the ellipse instead of the radius of the sphere:



```
for (var i = 0; i < slices; i++) {
    var angle1 = Math.PI - (i * Math.PI / slices);
    var nextAngle1 = Math.PI - ((i + 1) * Math.PI / slices);

    for (var j = 0; j <= slices; j++) {
        var angle2 = j * Math.PI / slices;

        var x1 = radius1 * Math.sin (angle1) * Math.cos (angle2);
        var y1 = radius1 * Math.sin (angle1) * Math.sin (angle2);
        var z1 = radius2 * Math.cos (angle1);

        var x2 = radius1 * Math.sin (nextAngle1) * Math.cos (angle2);
        var y2 = radius1 * Math.sin (nextAngle1) * Math.sin (angle2);
        var z2 = radius2 * Math.cos (nextAngle1);
```

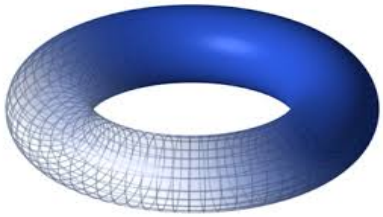
The head is translated continuously during the game, and the direction of the translation is based on the commands it recieves by the user.

The **body** of the snake is organized in cylinders. At the beginning of the game we only have two of them. The number of snake body parts increases every time an apple is eaten. Each snake body is built using cylindrical coordinates:

```
for (var i = 0; i < slices; i++) {
    var bottom = -(height / 2) + (i * height / slices);
    var top = -(height / 2) + ((i + 1) * height / slices);

    for (var j = 0; j <= slices; j++) {
        var angle = j * Math.PI / slices;
        var x = radius * Math.cos(angle);
        var y = radius * Math.sin(angle);

        var sideNorm = vec4(Math.cos(angle), Math.sin(angle), 0.0, 1.0);

        var v1 = vec4(x, y, bottom, 1.0);
        var v2 = vec4(x, y, top, 1.0);
```

The body parts have a particular animation: they don't really move, but their texture flows through them. To make the snake procede, at every moment a slice of the last body part before the tail disappears, while a new slice of the first body part after the head appears. When the new body part is complete, a new one is spawned right behind the head, and so on. This method helps improving efficency, because it reduces the number of parts which change position and for which vertices, and normals have to be recalculated.

During the rotation the body has to follow the rotation of the head. For this reason we created **SnakeLeftBody** and **SnakaRightBody**, which are built using toroidal coordinates :



x(p,t) = (R+r cos(a2))cos(a1)
y(p,t) = r sin(a2)
z(p,t) = (R+r cos(a2))cos(a1)

```
for (var i = 0; i < slices; i++) {
    var a1 = i * (Math.PI / 2) / slices;
    var nA1 = (i + 1) * (Math.PI / 2) / slices;

    for (var j = 0; j <= slices; j++) {
        var a2 = Math.PI - (j * Math.PI / slices);
        var nA2 = Math.PI - ((j + 1) * Math.PI / slices);

        var x1 = 0.5 -(0.5 - radius * Math.cos (a2)) * Math.cos (a1);
        var y1 = radius * Math.sin (a2);
        var z1 = -0.5 + (0.5 - radius * Math.cos (a2)) * Math.sin (a1);

        var x2 = 0.5 -(0.5 - radius * Math.cos (a2)) * Math.cos (nA1);
        var y2 = radius * Math.sin (a2);
        var z2 = -0.5 + (0.5 - radius * Math.cos (a2)) * Math.sin (nA1);

        var nX1 = 0.5 -(0.5 - radius * Math.cos (nA2)) * Math.cos (a1);
        var nY1 = radius * Math.sin (nA2);
        var nZ1 = -0.5 + (0.5 - radius * Math.cos (nA2)) * Math.sin (a1);

        var nX2 = 0.5 -(0.5 - radius * Math.cos (nA2)) * Math.cos (nA1);
        var nY2 = radius * Math.sin (nA2);
        var nZ2 = -0.5 + (0.5 - radius * Math.cos (nA2)) * Math.sin (nA1);

        var v1 = vec4 (x1, y1, z1, 1.0);
        var v2 = vec4 (x2, y2, z2, 1.0);

        var nV1 = vec4 (nX1, nY1, nZ1, 1.0);
        var nV2 = vec4 (nX2, nY2, nZ2, 1.0);
```

A quarter of a torus gives a curved cylinder, which exactly fits the necessities. These body parts follow the same animation logics of the rest of body parts.

The **tail**, like the head, is built using ellipsoidal coordinates, but this time it is represented by only half ellipsoid. Also the tail is continuously translated following the motion of the rest of the body. In practice head and tail are the only moving piecies of the snake.

The hierarchical structure as we have seen in the introduction is organized in a way that all the pieces follow the antecedent element (the parent) during the movement. In the next section we will see how the snake moves and how it stretches.

Sea

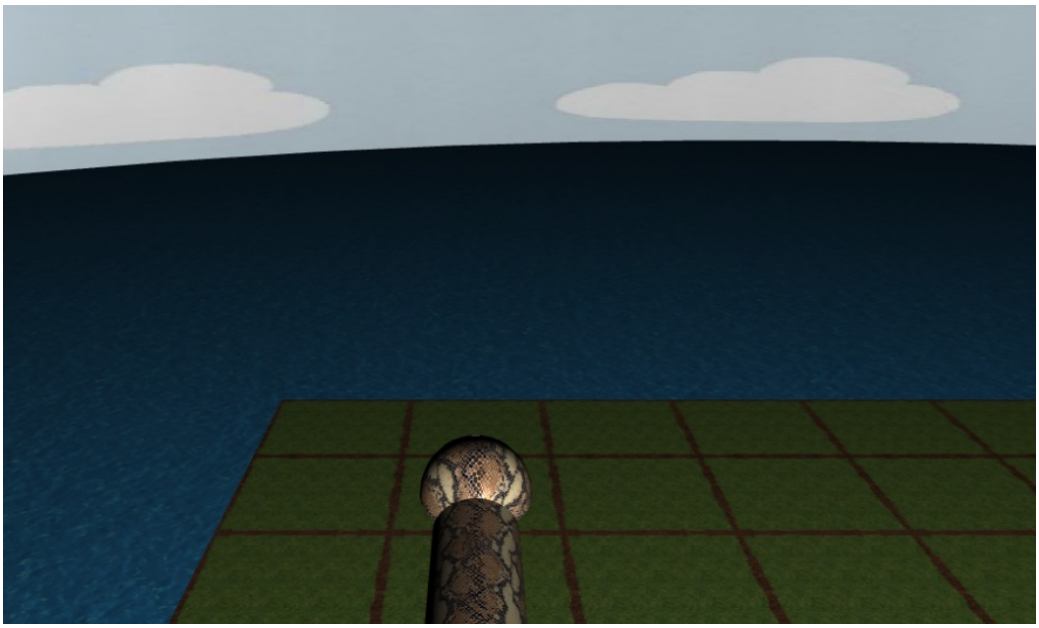
This object is a square which contains a texture of the sea. We simulated the movement of the waves of the sea animating this object by the use of sinusoidal functions:

```
this.move = function () {
	var delta = 0.01 * Math.sin(this.seaWave * Math.PI / 180.0);
	this.model = mult (this.model, translate (delta, 0.0, delta));
	this.modelNorm = normalMatrix(this.model, false);
	this.seaWave = (this.seaWave + 1) % 360;
};
```

Sky

This object is a cylinder which goes all around the other objects, and it has negated normals w.r.t. the other cylinders, so that lights and textures are applyed to its inner faces. We use the texture of a cloudy sky and we animate it by rotating it around its y axis, so that the clouds move accordingly:

```
this.move = function () {
	this.model = mult (this.model, rotateY (0.1));
	this.modelNorm = normalMatrix(this.model, false);
};
```

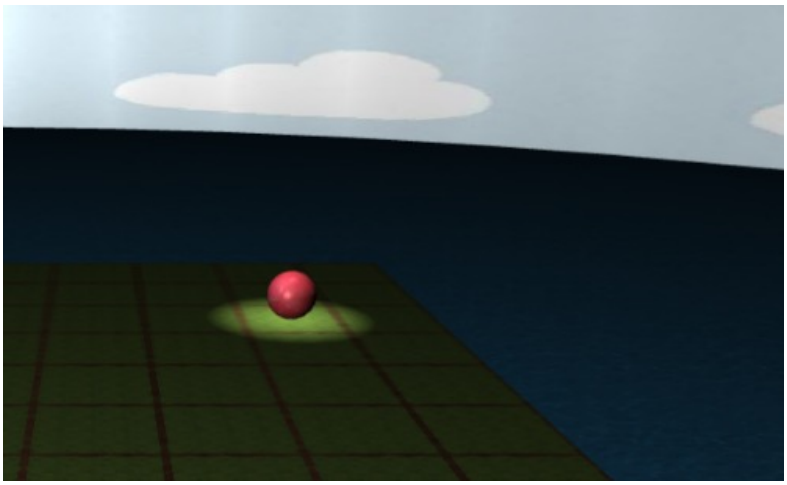


Lights

We have already said that for the lights we have used both Phong and Gouraud models depending on the distance of the elements by the camera. We also use two different kinds of lights:

- **Spotlight**
  To create a spotlight we define a cutoff angle, out of which the illumination is zero, so that the result is a cone of light. The spotlights are placed on the top of apples and illuminate in their direction, so only apples and few ground around them are enlightened.

- **Positional light**
  The positional light is centered on the top of the scene and acts on all the elements in the environment. The intensity of this light is more attenuated than that of the spotlight because it is further.



Interactions

In this section the most important element is the user interaction. In fact is the user that decide what the snake must do. We talk first about the general movement and than of what happens if the snake eats the fruit.

- **Reference Matrix**
  We create a matrix which has the same dimension of the world and which is filled at each iteration with the objects in the environment. In this matrix we trace the position of the bonus with the char 'b' and of the snake, indicating with the char 'h' the position of the head and with the char 's' the position of all the other components of the snake. This matrix is very important for drawing the mini map, understanding

if the lose conditions are reached and calculating the new position of the apple when it is eaten.
The position of each element is taken through the model matrix.

- **Movement with the user iteration**
  If the user doesn't press a key the snake goes straight. If the user press the space bar the movement of the snake is paused. The snake can be controlled using the arrow keys: pressing right or left arrow on the keyboard makes the snake turn in the respective direction.
  As we said before, the snake can only occupy certain positions in the world, because it has to move accordingly to the grid, so if the user presses a key, the command is executed when the head of the snake is at the beginning of the next cell, and then, accordingly to the key pressed, a new body part is spawned: a straight cylinder if no key is pressed, a left-curved cylinder if left arrow key is pressed and a right-curved one if right arrow key is pressed. The head immediately starts rotating in the appropriate direction, while the tail begins rotating when it reaches the point where the animation started for the head.

```
document.onkeypress = function(e){
    e = e || window.event;
    var code = e.keyCode;
    if (code === 37) { // <-
        if (!Snake.turningRight) Snake.turningLeft = true;
        return;
    }
    if (code === 39) { // ->
        if (!Snake.turningLeft) Snake.turningRight = true;
        return;
    }
};
```

- **Eating apples**
  When the snake head is in the same cell of an apple, it disappears and the snake stretches by one element. To calculate the new position of the apple we choose a random position in the world which is not occupied by the snake. As we said, when the snake eat the apple we must insert a new piece of body, so a new cylinder. To do this in an animated way we exploited the hierarchical structure and the movement logics of the snake: the head continues translating normally and the first cylinder behind it follows drowing its slices, while the tail stops translating and the last cylinder before it stops disappearing until the end of the animation. There is also a counter which indicates the number of apples eaten.

- **Win or Game Over**

The winner status is reached when the counter equals a certain number of apples. This number is based on how long can the snake become long before it occupies every cell in the world.
The loser status is reached when the snake goes out of the world or it collides with itself. These conditions are calculated through the model matrices of the objects and the reference matrix: if the head of the snake is in the same cell with another snake body part, or it is in a cell out of the grid, we send the game over message. In both cases the user can choose to restart the game and the page is refreshed.

- **Mini Map**
  The mini map is a canvas added to the main window to help the user finding the apple in relation to its position in the world. We have divided the canvas in cells like the world, and through the reference matrix we have represented the apple in red and the head and the first cylinder in grey. The empty cells of the world are black. The mini map rotates in order to have on the top side the direction in which the snake is going, because since the camera rotates following the snake, also the mini map has to adjust, otherwise it would be even more confusing than not having it at all. In order to do so, at each iteration we rotate the canvas following the direction of the snake.