Underwater World

- Graphical Processing Systems Project -

Petrican Teodor
Gr. 30433
Technical University of Cluj-Napoca

1. Contents

2. Subject specification

The subject of the project consists of creating an application which renders a realistic 3D scene of objects using the OpenGL API. The 3D scene must contain several objects and exemplify techniques such as lighting, shadows, animations, fog, user interaction (mouse and keyboard) etc. The program language in which the project was developed is C++.

The scene selected for this project, as it can be seen in the title, is an underwater world. A more detailed explanation about the scene is presented in the following section.

3. Scenario

3.1. Scene and objects description

As previously stated, the scene depicted is an underwater world. To be more specific, the scene is that of a shallow sea, which contains a sunken ship on the bottom, several sea plants and some fauna (fishes of various species and starfishes), as it can be seen in Fig. 1 below.

The fishes swim in all directions give life to the whole scene. There are also air bubbles which are rising to the surface of the water. To achieve a realistic feeling of being under water, there is present the optical effect known as *caustics*. In optics, a caustic is the envelope of light rays reflected or refracted by a curved surface or object, or the projection of that envelope of rays on another surface [1].

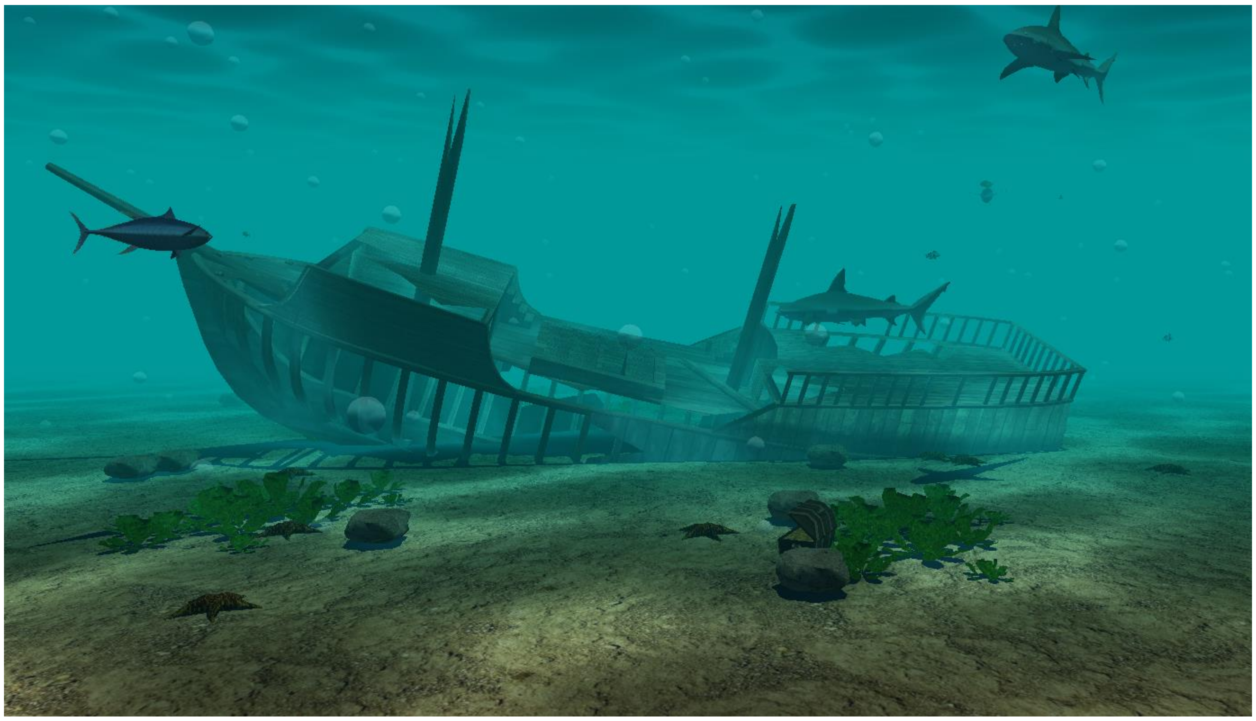


Fig. 1 – screenshot of the scene depicted in the application

The scene contains several types of objects, which include the following:

- Fishes: sharks, tunas, and several other exotic fishes;
- Starfishes, which are laid on the bottom of the sea;
- A sunken ship;
- A sunken gold chest;
- Rocks and plants, all of the same type.

3.2. Functionalities

The user can interact with the application in the several ways, presented below.

3.2.1. Camera movement

The most important functionality is the control of the "player" which looks at the scene in a first-person manner (the user controls directly the camera). The camera is controlled via the mouse and the keyboard: the mouse is used to rotate the view in any direction desired and the W, A, S, D keys are used to move the camera around in the scene.

The mouse pointer is hidden and trapped within the window frame.

3.2.2. Scene manipulation

The user can also choose several options to manipulate the scene (e.g. enable/disable caustics, enable/disable air bubbles etc.), which will directly affect the way in which the scene is rendered.

Via this menu the user can also choose the type of shadows displayed: projected shadows, volumetric shadows, shadow volume or no shadows at all.

These options can be accessed via a menu on the right side of the application. Please refer to the *GUI Presentation* section of this documentation for further details about accessing the Scene manipulation menu.

3.3. Features

The application offers a number of features, which refer directly to the 3D scene rendering, out of which the most important are: collision detection, caustics, random fish directions, projected shadows, volumetric shadows exemplification and the possibility to go out of the water. Most of these features will be described in detail in the next section.

3.3.1. Collision detection

The camera cannot be moved forward into objects from the scene (i.e. it collides with the objects). Whenever the camera hits another object, it will stop is possibility to move in that direction. This is true for any object except the air bubbles.

4. Implementation details

4.1. Functions and special algorithms

In this subsection, several functions of the application and their underlying principles are presented. Some small pseudo-code snippets are presented. Please note that they are just to better visualize the principles and are not a 1-to-1 representation of the actual code of the application (they are just resembling the idea of the actual implementation). For the actual implementation of the details described here please refer to the code of the project.

4.1.1. Caustics

The caustics have an important role in adding a touch of realism to the whole scene. The application simulates the way in which light behaves underwater in a real case scenario. The caustics are present on all objects of the scene, but they are mainly evident on the bottom and top of the sea.

The technique used to generate the caustics involves a second rendering pass over all objects in the scene, but this time texturing objects with a texture as the ones in Fig. 2 and using a blending function, in order to blend the texture with the previous render of each respective object.

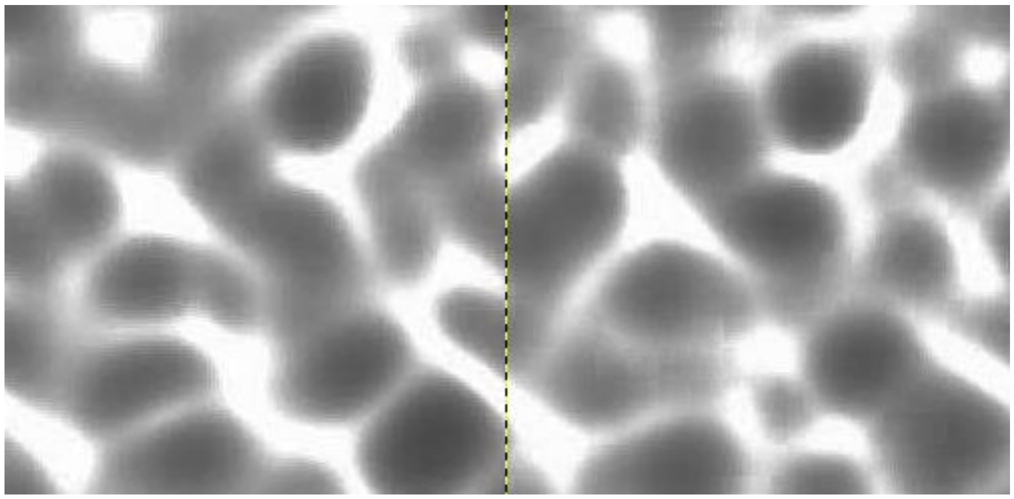


Fig. 2 – example of 2 caustics frames

As it can be seen, these textures are grayscale. The blending is done such that the white is fully transparent, and it increases in opacity as the intensity of the texture increases.

There are 32 caustic textures, which are looped through in order to create the animation effect. The caustic textures were generated in such a way that they will look seamless when added next to each other, and that they will provide a continuous, smooth animation.

The pseudo-code presented below tries to capture the idea explained above. The rendering of a scene consists of two passes, one normal pass and one with the caustic texture. There is an active caustic which is changed by an independent loop (a loop calling setActiveCaustic()), in order to create animation. The active caustic is used in the caustic pass to achieve the desired effect.

```
function renderScene()
 begin
       drawScene(PASS_NORMAL);
       drawScene(PASS_CAUSTIC);
 end;

 active_caustic;

 function setActiveCaustic()
 begin
         static int i = 0;
         active_caustic = caustics[i];
         i=i+1;
 end;

 function drawScene(pass)
 begin
 …
 if (pass==PASS_CAUSTIC)
         setTexture(active_caustic);
 …
 end;
```

By controlling the frequency of the frames (which is straightforward – frequency of calling setActiveCaustic()), we can manipulate the speed of caustics animation. Also, the size can be easily controlled. In this project, variables have been set from which the size and speed of the caustics can be altered.

This technique is presented on the OpenGL website [2], along with an example implementation. Also, links to pages regarding the caustic textures generation are given. All credits for this technique go to Mark Kilgard (author of the tutorial), Jos Stam and Angus Dorbie (developers of the technique).

4.1.2. Fishes movement

The fishes in the scene are moving in random directions. They are also rotating to face the direction in which they are going. The implementation of this is similar to the following pseudo-code:

```
function moveFish()
begin
      direction_vector = target - fish_position;
      makeOneStep();
      if (fish_position==target)
            generateOtherTarget();
      rotateFishToDirection()
end;
```

The fishes start at random positions and with a random target. Each time their move function is called, they will make on step toward the target position, on the direction vector. The length of a step may vary, depending on the fish speed. If the target position has been reached, another target is chosen randomly, and the process continues.

They will always rotate to match the direction of movement. This can be achieved by simple trigonometry required to find the angle between 2 vectors.

After rotating towards the direction of movement, they also perform an additional rotation by an angle between -10 and 10 degrees (changing each time), to better simulate the way fishes swim.

4.1.3. Collision detection

In order to add a sense of presence of the objects in the scene, a collision detection mechanism has been implemented. The user cannot move forward the camera through objects in the scene. Whenever the camera hits an object, the user will be unable to continue moving the camera in that direction. The mechanism implemented for the collision detection is somewhat simple, though it has drawbacks.

The mechanism implies reading the depth buffer after each rendering of the scene (after each frame), and checking if there are values less than 0.5 in the depth buffer. If there are, it means that there are objects in the scene which are very close the camera (from the way the depth buffer works) and the movement needs to be blocked. This will ensure a fast collision mechanism with a very good precision: it will perfectly collide with objects of complex shapes (e.g. the sunken ship).

The major drawbacks of this technique are that it cannot check for collision when moving backwards (since no information *behind* the camera is collected by the depth buffer during rendering) and that it works only for the collision of the camera with other objects, and not for anything else.

4.2. Graphics model

4.2.1. Modelling – tools used

For 3D modeling and objects manipulation, Blender was used. Blender is an open-source 3D graphics software product [3]. It was the product of choice because it is free, there was previous experience with the program and it suited well the needs. For textures, GIMP was used, which is another open-source software used for photo manipulations.

The 3D models were exported in .obj format which was used in the application. The UV Unwrapping technique was applied directly from Blender, so that the textures are mapped correctly on the objects when imported in the application.

In order to import the objects, the GLM library was used, which provides all the functionality required to load .obj models into the application. The textures were saved in the .tga (Targa) format and were imported into the application using another library, “tga.h”.

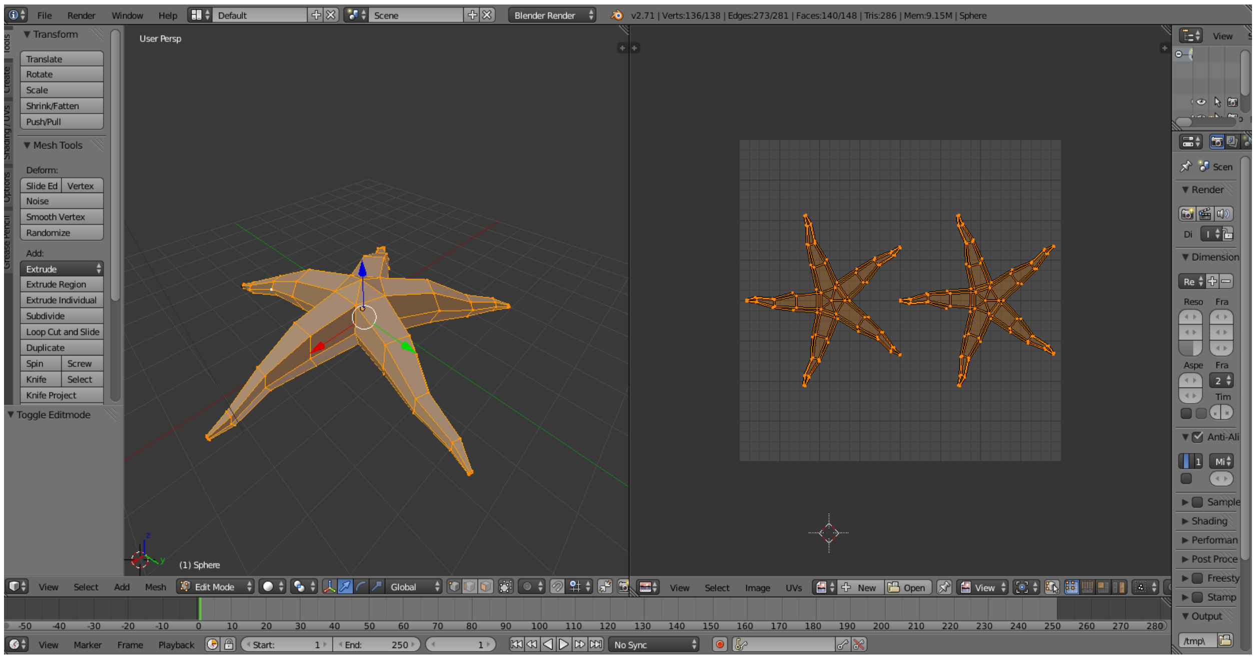

v2.71 | Verts:136/138 | Edges:273/281 | Faces:140/148 | Tris:286 | Mem:9.15M | Sphere

Fig. 3 – example of modeling and UV unwrapping in Blender of the starfish

4.2.2. Fog

The scene includes fog, which is needed to simulate the presence of water, since an underwater view is very similar to the one on a foggy day, except that the fog is blue. Therefore a blue fog is rendered.

To generate fog, a simple mechanism is used, which is built-in in OpenGL. The usage of fog can simply be enabled by calling glEnable(GL_FOG). When fog is enabled, objects that are father from the viewpoint begin to fade into the fog color. The density of the fog can be controlled, as well as the fog's color. [4]

In this case, the fog's color is a blue nuance. The fog density function is set to GL_EXP2 with a density factor of 0.01.

$$f = e^{-(density.z)} \quad \text{(GL\_EXP)}$$
$$f = e^{-(density.z)^2} \quad \text{(GL\_EXP2)}$$
$$f = \frac{end - z}{end - start} \quad \text{(GL\_LINEAR)}$$

Fig. 4 – fog density equations [4]

4.2.3. Light sources

In this project there are two sources of light. One of them is a global, directional light source, which has a hint of blue, in order to emulate lighting underwater. It is coming from outside of the water. The second light source is a red spotlight, coming from a lantern and placed near the ship. The red spotlight will vary in intensity and its effects can be seen on the back of the ship.

4.2.4. Transparent air bubbles

The scene also contains transparent air bubbles. The bubbles are spherical objects which are rendered without texture and with a transparent material (alpha component set to less than 1). Then, the blending mechanism offered by OpenGL is used to make the bubbles transparent. The blending function used is glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA). For more information regarding how blending works in OpenGL, please refer to the OpenGL Red Book [4], or similar sources.

4.2.5. Projected Shadows

The scene features shadows. One of the types of shadows rendered is known as projected shadows. The name comes from the fact that the shadow is actually the object (of whose shadow is drawn) projected onto a plane (in this case, the bottom of the sea) and drawn in full black color.

This method is very fast, since it involves only the computation of a "shadow matrix" (projection of object on a plane, which is dependent on the position of the light and the plane) and a second drawing of each object, multiplying the Modelview matrix with the "shadow matrix". This way, each vertex of the object will be projected on the plane on which the shadow is cast, along the light ray.

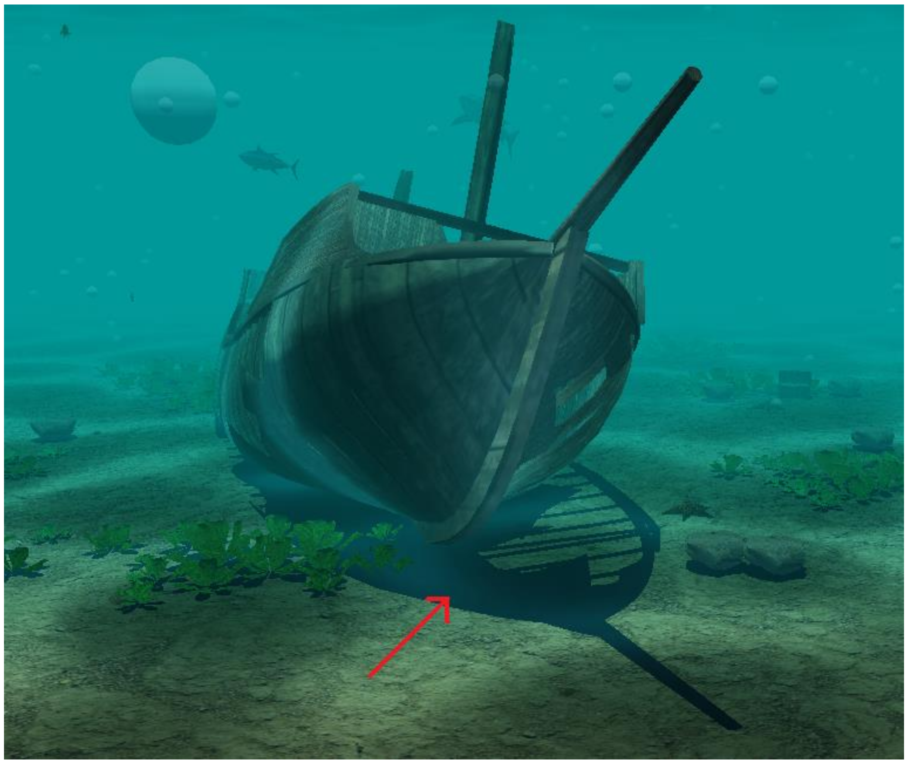


Fig. 5 – projected shadow of the ship on the bottom of the sea

One drawback of this method of generating shadows is that the shadow can only be (easily) projected onto planes, and therefore it will not map onto more complex objects. As it can also be seen in Fig. 5 above, the sea plants are not affected by the ship's shadow.

One problem encountered (strictly related to implementation) was that, by applying the function glMultMatrix(shadowMatrix) in OpenGL, the fog calculations were affected, and the shadow appeared with maximum fog color instead of appearing with their designated color. This is because fog calculations use the modelview matrix. The only way to avoid this was to move the multiplication with the modelview matrix in the projection matrix, while keeping the same order of operations.

The normal operations pipeline is as follows:

```
    Shadow = ProjectionMatrix * ModelviewMatrix * ShadowMatrix *
Object
```

The idea is to have a ProjectionMatrix2, s.t.:

```
    ProjectionMatrix2 * ModelviewMatrix * Object = ProjectionMatrix *
ModelviewMatrix * ShadowMatrix * Object
```

⇨ `ProjectionMatrix2 = ProjectionMatrix * ModelViewMatrix * ShadowMatrix * ModelviewMatrix`$^{-1}$

This way, having:

```
    Shadow = ProjectionMatrix2 * ModelviewMatrix * Object
```

will have the same effect as the normal operations order, except that the multiplication with the shadow matrix is not done in modelview, but in projection and fog is no longer affected.

4.2.6. Volumetric Shadows

Another type of shadows is the volumetric shadows. Their name comes from the fact that a shadow volume is generated, and all objects that are inside the shadow volume are drawn without lighting (are not affected by light, are in shadow). The shadow volume is basically a volume into which the light cannot enter because it is obstructed by an object.

The generation of the shadow volume is, hence, generated taking into consideration the position of the light, and the object(s) which casts shadows. Fig. 6 presents the idea of shadow volume generation.

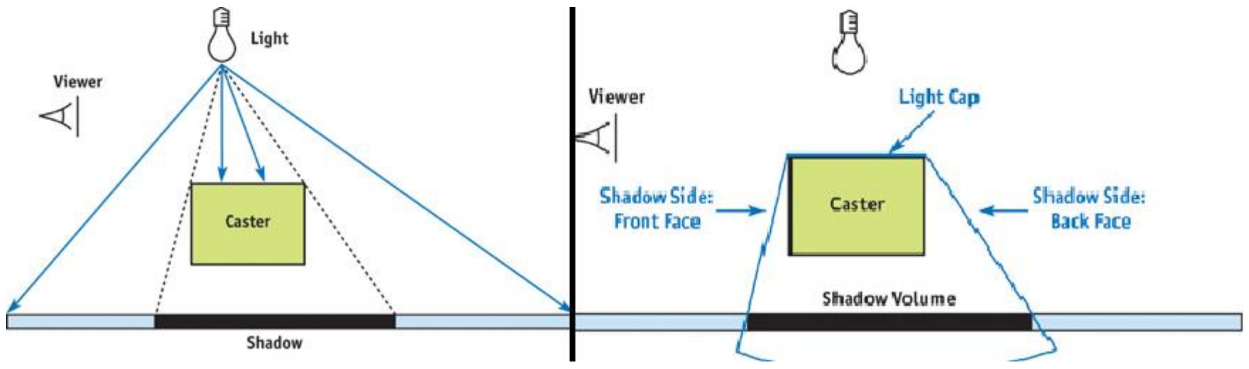


Fig. 6 – shadow volume [5]

The shadow volume is extended, from the caster, along the light rays up to infinity (actually, a distance which is convenient). The shape which has to be extended to generate the shadow volume is represented by the *silhouette* of the casting object, as seen from the light source.

The algorithm implemented in this project to determine the silhouette of the caster object checks all the edges of the object, and verifies the two faces to which the edge belong. If one of the faces is front-facing the light, and one is back-facing, then the edge belongs to the silhouette (Fig. 7).

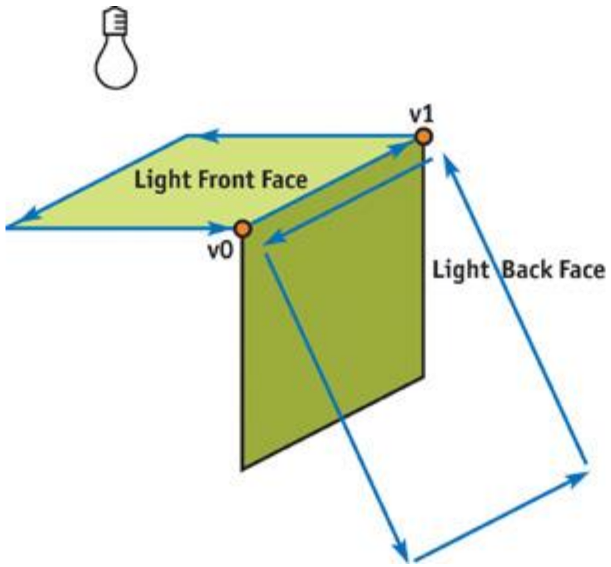


Fig. 7 – silhouette generation [5]

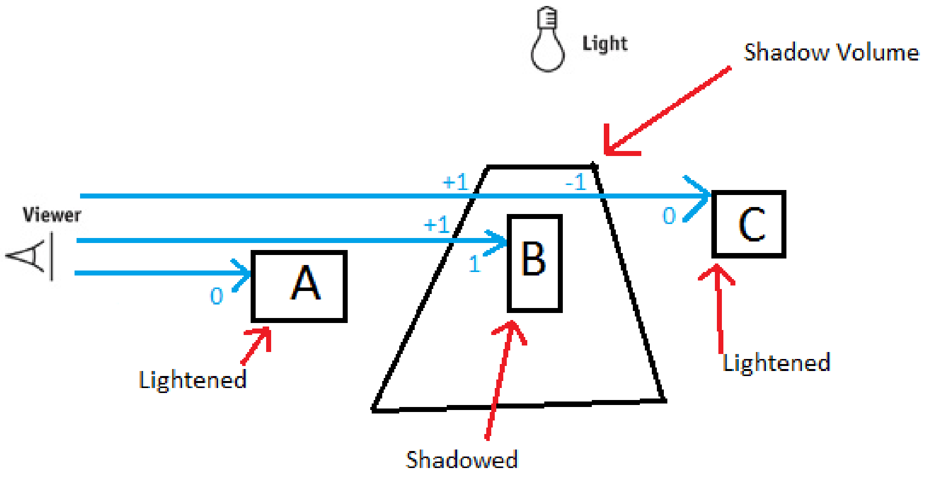


Fig. 8 – calculation of objects affected by the shadow volume

Once the shadow volume is generated, the next step of the algorithm can take place: drawing the scene, with the objects falling inside the shadow volume being shadowed. To achieve this, the OpenGL stencil buffer is used. The shadow volume is drawn using back-face culling, and incrementing the stencil buffer. Now we will have positive values in the stencil buffer wherever front faces of the shadow volume were drawn. The shadow volume is drawn a second time, with front-face culling, and decrementing the stencil buffer. Now, we will have values of 0 in the stencil buffer wherever pixels correspond to objects that are outside of the viewing volume, and different from 0 wherever pixels correspond to objects that are inside of the viewing volume. The illustration of the mechanism is presented in Fig. 8, above.

In this project the volumetric shadows mechanism is exemplified on only one object, which is a boat which floats to the surface of the water. The boat casts shadows on the sunken ship below. There is a mode in which the shadow volume can be seen, and a mode in which the volumetric shadow is rendered. Those can be changed from the application menu (see section 5 – User manual).

4.2.7. Water surface – Skydome

There is the possibility of leaving the water through the top and reaching the water surface. This is not intended to be a very realistic depiction of the water surface, since this is not the main scope of the application.

Instead, this component was added more to illustrate the concept of skydome. The skydome is very similar to the more-known skybox method of creating the “background” for a scene. The whole difference between the skybox [6] and the skydome, as the name suggests, is that the skydome uses a hemisphere instead of a cube, onto which the texture is mapped.

The water surface color and the sky color do not blend very well together, but, again, it was not intended to be a very realistic depiction of the water surface.

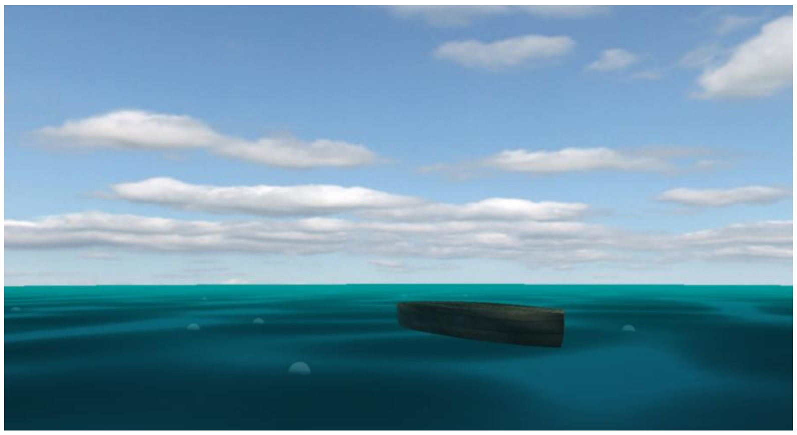


Fig. 9 – water surfer and skydome

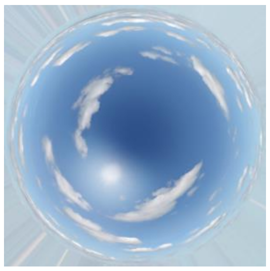
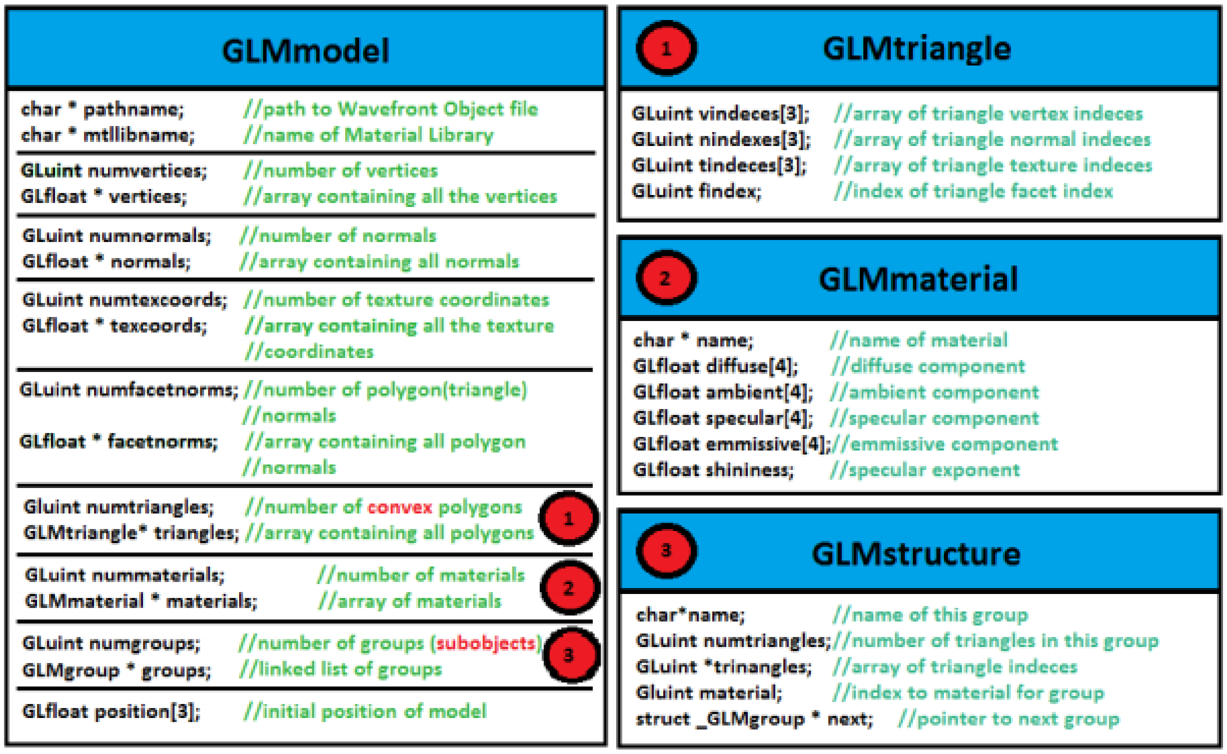


Fig. 10 – skydome texture

4.3. Data structures

An important data structure is the GLMmodel, offered by the GLM library, into which the .obj model is stored. The GLMmodel structure looks like in the figure below.



GLMmodel

```
char * pathname;          //path to Wavefront Object file
char * mtllibname;        //name of Material Library

GLuint numvertices;       //number of vertices
GLfloat * vertices;       //array containing all the vertices

GLuint numnormals;        //number of normals
GLfloat * normals;        //array containing all normals

GLuint numtexcoords;      //number of texture coordinates
GLfloat * texcoords;      //array containing all the texture
                          //coordinates

GLuint numfacetnorms;     //number of polygon(triangle)
                          //normals
GLfloat * facetnorms;     //array containing all polygon
                          //normals

Gluint numtriangles;      //number of convex polygons          1
GLMtriangle* triangles;   //array containing all polygons

GLuint nummaterials;           //number of materials           2
GLMmaterial * materials;       //array of materials

GLuint numgroups;         //number of groups (subobjects)      3
GLMgroup * groups;        //linked list of groups

GLfloat position[3];      //initial position of model
```

1 GLMtriangle

```
GLuint vindeces[3];   //array of triangle vertex indeces
GLuint nindexes[3];   //array of triangle normal indeces
GLuint tindeces[3];   //array of triangle texture indeces
GLuint findex;        //index of triangle facet index
```

2 GLMmaterial

```
char * name;            //name of material
GLfloat diffuse[4];     //diffuse component
GLfloat ambient[4];     //ambient component
GLfloat specular[4];    //specular component
GLfloat emmissive[4];   //emmissive component
GLfloat shininess;      //specular exponent
```

3 GLMstructure

```
char*name;              //name of this group
GLuint numtriangles;    //number of triangles in this group
GLuint *trinangles;     //array of triangle indeces
Gluint material;        //index to material for group
struct _GLMgroup * next;    //pointer to next group
```

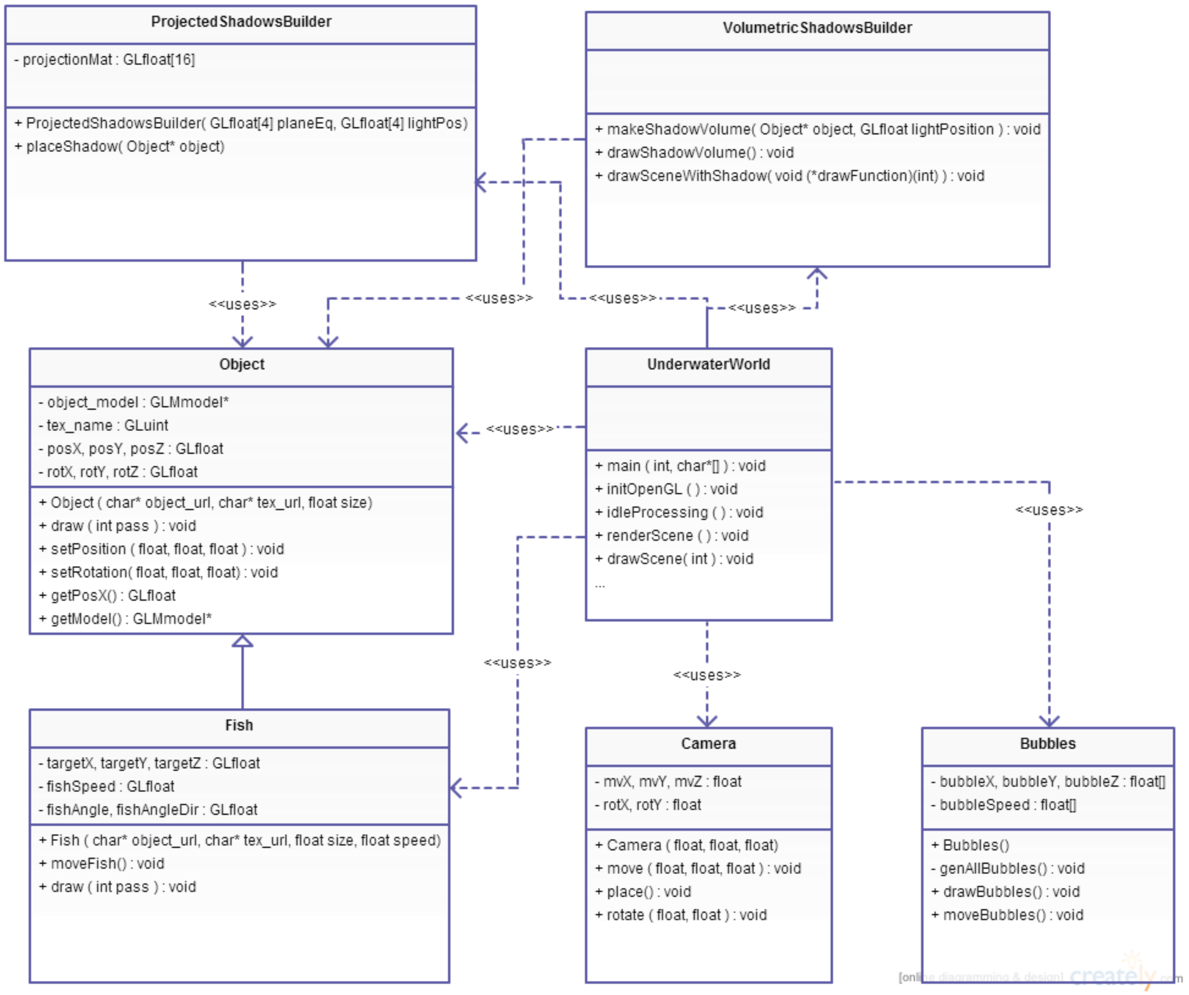
Fig. 11 – GLMmodel structure [7]

There are not many others data structures worth noting. Most of the information needed for computation (coordinates, speed, etc.) was stored in simple variables or in simple arrays.

4.4. Class hierarchy

The project is developed in C++ and is split in several classes to provide an elegant solution. The class diagram of the project can be seen in Fig. 12 below. It can be seen from the class diagram that UnderwaterWorld contains the entry point for the application, and uses all other classes of the project. Also, in UnderwaterWorld.cpp, the entire interface is created and most of the rendering computation is done.

One thing to note is the relationship between the Object class and the Fish class. The Object class allows the creation of a 3D object (the class automatically imports the object and its texture upon construction), and allows for positioning it and drawing it (provides methods for this). The Fish class extends the Object class' functionality (since fish is also an object), but further adds functionality to move the fish (moveFish() – deals with moving and rotating the fish to the right position/direction).


ProjectedShadowsBuilder
- projectionMat : GLfloat[16]
+ ProjectedShadowsBuilder(GLfloat[4] planeEq, GLfloat[4] lightPos)
+ placeShadow(Object* object)
VolumetricShadowsBuilder
+ makeShadowVolume(Object* object, GLfloat lightPosition) : void
+ drawShadowVolume() : void
+ drawSceneWithShadow(void (*drawFunction)(int)) : void
<<uses>>
<<uses>>
<<uses>>
<<uses>>
Object
- object_model : GLMmodel*
- tex_name : GLuint
- posX, posY, posZ : GLfloat
- rotX, rotY, rotZ : GLfloat
+ Object (char* object_url, char* tex_url, float size)
+ draw (int pass) : void
+ setPosition (float, float, float) : void
+ setRotation(float, float, float) : void
+ getPosX() : GLfloat
+ getModel() : GLMmodel*
<<uses>>
UnderwaterWorld
+ main (int, char*[]) : void
+ initOpenGL ( ) : void
+ idleProcessing ( ) : void
+ renderScene ( ) : void
+ drawScene(int) : void
...
<<uses>>
<<uses>>
<<uses>>
Fish
- targetX, targetY, targetZ : GLfloat
- fishSpeed : GLfloat
- fishAngle, fishAngleDir : GLfloat
+ Fish (char* object_url, char* tex_url, float size, float speed)
+ moveFish() : void
+ draw (int pass) : void
Camera
- mvX, mvY, mvZ : float
- rotX, rotY : float
+ Camera (float, float, float)
+ move (float, float, float) : void
+ place() : void
+ rotate (float, float) : void
Bubbles
- bubbleX, bubbleY, bubbleZ : float[]
- bubbleSpeed : float[]
+ Bubbles()
- genAllBubbles() : void
+ drawBubbles() : void
+ moveBubbles() : void
[online diagramming & design] creately.com
Fig. 12 – class diagram

The Bubbles class provides functions for bubbles drawing and bubbles moving (similar to the Fish class, but simpler). The Camera class implements all the camera movement functionalities, which were presented in the previous section (move, rotate, place i.e. apply the movement and rotation).

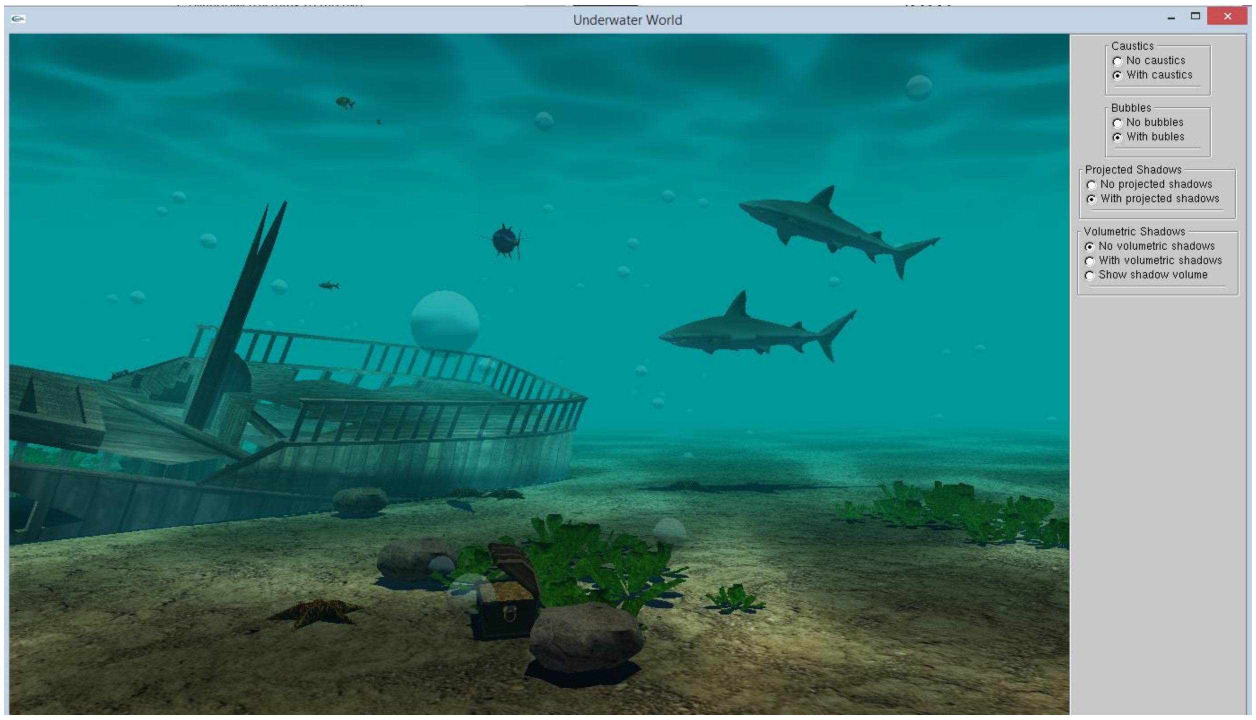
The ProjectedShadowsBuilder class computes the projection matrix directly from the constructor. It then provides the placeShadow() function, which gets a pointer to the Object of which shadow needs to be drawn as an argument. Since the given Object is also characterized by position and rotation, those are directly applied on the projected shadow.

The VolumetricShadowsBuilder class provides methods for the creation of the shadow volume, for drawing the shadow volume, or for rendering the scene with volumetric shadows. For the latter, the function takes as an argument a pointer to the scene rendering function. In this case, the drawScene() function from UnderwaterWorld is given.

5. User manual

5.1. Main Window

The application consists of only one window, onto which the scene is visualized and which also has a menu available on the right side. By default, the mouse pointer is hidden and trapped inside the window. This is necessary to achieve the camera rotation by mouse.



Caustics: No caustics [✓ With caustics]

Bubbles: No bubbles [✓ With bubles]

Projected Shadows: No projected shadows [✓ With projected shadows]

Volumetric Shadows: [✓ No volumetric shadows] With volumetric shadows Show shadow volume

Fig. 13 – main window of the application

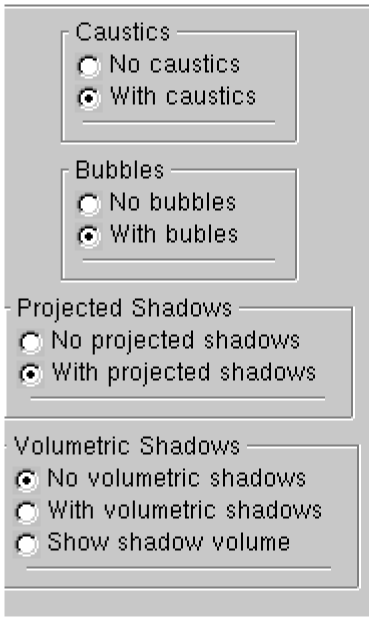
5.2. Controls

The controls of the application are the following:

- Camera:
  - Rotate in all directions: move mouse
  - Move forward: W
  - Move backward: S
  - Move left: A
  - Move right: D
- Toggle escape/trap mouse pointer: M
- Exit application: Q

5.3. Menu

The application offers a simple menu which is present on the right side of the window. The menu offers options which affect the scene rendering, like caustics enable/disable, bubbles rendering enable/disable or choosing the type of shadows that are rendered.



Caustics
No caustics [✓ With caustics]

Bubbles
No bubbles [✓ With bubles]

Projected Shadows
No projected shadows [✓ With projected shadows]

Volumetric Shadows
[✓ No volumetric shadows] With volumetric shadows Show shadow volume

Fig. 14 – application menu close-up

To be able to access the menu, first press the M key in order to escape the mouse pointer. Navigate with the mouse to the menu, make appropriate choices, and then press M again in order to trap back the mouse pointer and regain camera control.

6. Conclusions and further developments

6.1. Conclusions

The idea chosen for the scene in this project was a good choice, since it dealt with many graphics techniques and aspects of the computer graphics development and the final result is pleasing.

The whole process of developing this project represents a very good training on many aspects related to computer graphics. The first and most important thing learned, was how to use OpenGL to render a 3D scene of objects. In OpenGL many techniques were studied, ranging from objects manipulation in 3D space to some advanced techniques for realistic effects. Out of the techniques involved in the development of this application I would like to mention: getting around the OpenGL workflow, manipulating geometry (scaling, translations, rotations), creating simple animations, applying fog, rendering transparent objects, applying caustics, rendering projected shadows and volumetric shadows etc.

6.2. Further developments

There are a number of enhancements which could and be made to the application, and also some features that are not yet present, but could be added in the future and would represent a plus to the application. Out of those, some important ones are:

- More realistic rendering of the surface of the water part. This is a nice addition to the scene and would be a plus to try and render a more realistic water surface scene.
- Fishes to make smooth turns when changing directions. Right now fishes take sharp turns when changing directions; this is unrealistic. Their trajectories could be worked on, so that they follow curved paths.
- Objects (fishes) collision. Right now fishes may pass through each other, since there is no object collision. A simple collision detection mechanism like axis-aligned bounding-boxes could be implemented.
- Bump mapping to water surface and floor. The bump mapping technique is known to generate very good results in terms of realism. Its application on this project could be very useful for the surface and bottom of the sea, such they do not look flat like they do.
- Fishes animation. The animation of the fishes is simple, using only direct mathematical transformations (rotations, translations). The fishes do not move flaps and do not change the shape of their body. Adding a mechanism which allows for such an animation would certainly be a plus.

Some of these features are easier to implement, some are harder, but all of them involve the study of helpful and interesting techniques and mechanisms.

Bibliography

1. Caustic (optics). *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Caustic_%28optics%29.

2. Rendering of Underwater Caustics. *OpenGL.* [Online] https://www.opengl.org/archives/resources/code/samples/mjktips/caustics/.

3. Blender. *Blender.* [Online] http://www.blender.org/.

4. Blending, Antialiasing, Fog, and Polygon Offset. *GLProgramming - RedBook.* [Online] http://www.glprogramming.com/red/chapter06.html#name3.

5. Efficient Volume Shadows Rendering. *nvidia - GPUGems.* [Online] http://http.developer.nvidia.com/GPUGems/gpugems_ch09.html.

6. Skybox. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Skybox_(video_games).

7. Laboratory 5 - GLM. *GPS Lab Guide.*