Alexandria University
Faculty of Engineering
Computer and Systems Engineering Department
Fourth Year

Computer Graphics
Project Report

World of Goo 3D

A HIGHLY INTERACTIVE PUZZLE GAME

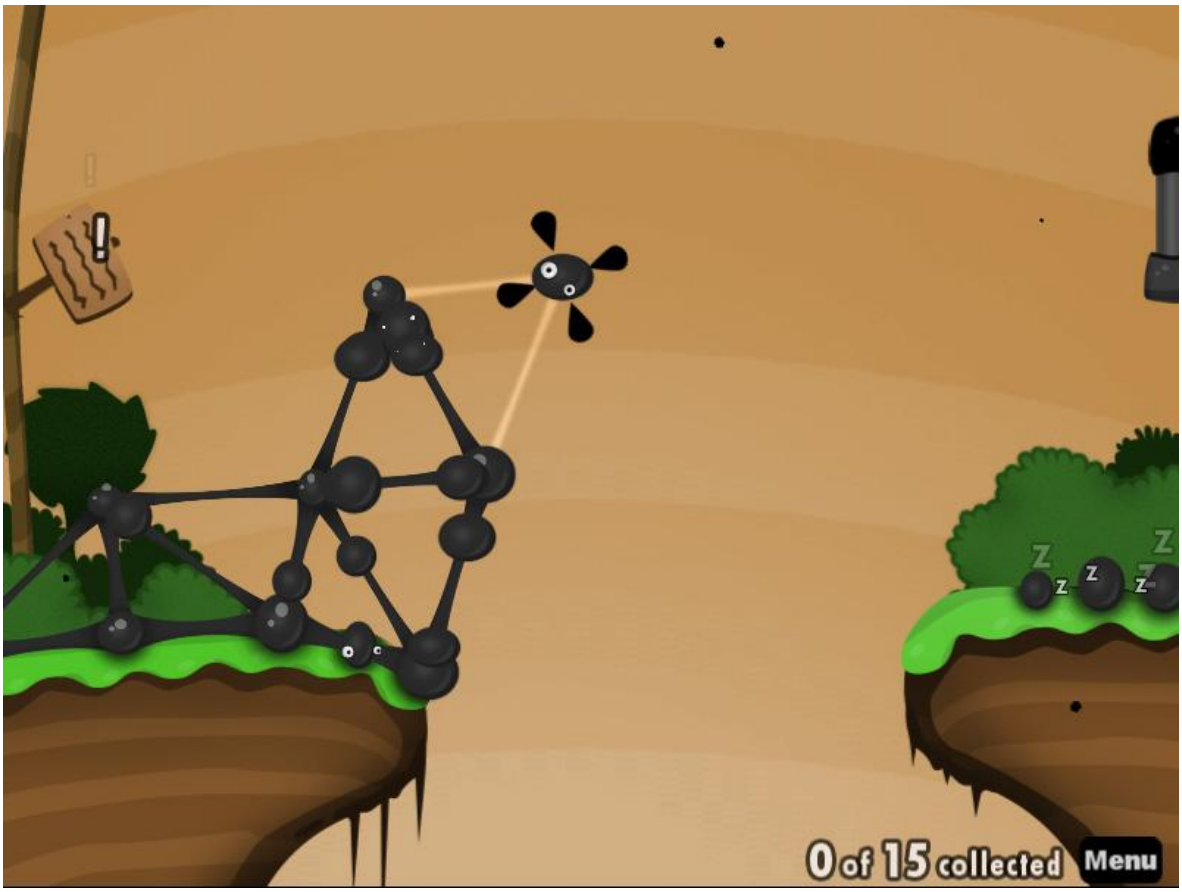
Presented by:

1. **Ahmed Ibrahim Elerian** **(1)**
2. **Ahmed Abdelkader Ramadan** **(8)**
3. **Ahmed Essam Mahmoud** **(10)**
4. **Ahmed Mahmoud Ahmed Mahran** **(15)**
5. **Mohamed El-Sayed Abdelfattah** **(49)**

1. Original Proposal

1.1 Overview

World of goo is a puzzle computer game, with a strong emphasis on physics, by 2D Boy. The game is built around the idea of creating large structures using balls of goo. The game is divided into five chapters, each containing several levels. Each level has its own graphic and musical theme, giving it unique atmosphere.



The whole game works in only 2 dimensions. We are planning to make another version that works in 3 dimensions. This will introduce new challenges to the player as well as the developer.

1.2 Gameplay

The main objective of the game is to get a requisite number of goo balls to a pipe representing the exit. In order to do so, the player must use the goo balls to construct bridges, towers and other structures to overcome gravity and various terrain difficulties such as chasms, hills, spikes or cliffs. There are several types of goo balls in the game, each of which has unique properties. The player must exploit combinations of these goo balls in order to complete each level.

1.3 Challenges

We will mainly have 3 types of challenges:

1. Physics

   We will need to implement gravity and elastic joints in 3D. We may also include other physics effects like wind for example. This will require a good deal of physics programming.

2. User Interaction

   The user should still be able to manipulate the goo structure easily. When extended to 3D, the user will need to look at the structure from different angles and place his goo balls in the 3D space to make his move. It will take some work to overcome the inherent complexity of the 3D model and make the game enjoyable.

3. Special Effects

   Without effects, the game would look dull. We will need to add visual and sound effects to enhance the gaming experience. We are planning to cover lighting in addition to a number of sound effects for each action.

1.4 Initial Plan

We will start by looking for libraries that could help us with the development of the game. However, we prefer to implement some parts ourselves. According to the available resources, we will determine the exact set of features that we are going to implement. After that, we will create an initial time plan and assign tasks among the team members. Hopefully, we will be able to have the game in good shape before the deadline.

2. Tools & Libraries

2.1 GLUT

GLUT, short for OpenGL Utility Toolkit, is a set of support libraries available on every major platform. OpenGL does not directly support any form of windowing, menus, or input. That's where GLUT comes in. It provides basic functionality in all of those areas, while remaining platform independent, so that you can easily move GLUT-based applications from, for example, Windows to UNIX with few, if any, changes.

2.2 SDL

Simple DirectMedia Layer is a cross-platform multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video frame buffer. In our project, we use this library specifically for sound effects.

We developed some methods to manage the sounds in our game. The methods provide utilities for controlling sounds during the game with high flexibility. Here are the interfaces we developed for sound management.

```
int SM_init(char* path);
int SM_terminate(int);

int SM_startPlay(int key, bool repeat, float vol);

void SM_stop(int slotIndex);
void SM_pause(int slotIndex);
void SM_resume(int slotIndex);
void SM_changeSoundVol(int slotIndex, float vol);

void SM_stopByKey(int key);
void SM_pauseByKey(int key);
void SM_resumeByKey(int key);
void SM_changeSoundVolByKey(int key, float vol);

void SM_stopAll();
void SM_pauseAll();
void SM_resumeAll();

void SM_mute();
void SM_unmute();
bool SM_isMuted();

bool SM_isActiveByKey(int key);
int SM_getState(int slotIndex);
void SM_setVolume(float vol);
```

```
float SM_getVolume();
void SM_setFreq(int freq);
int SM_getFreq();
void SM_setChannels(int channels);
int SM_getChannels();
```

2.3 ODE

The Open Dynamics Engine (ODE) is a free, industrial quality library for simulating articulated rigid body dynamics. For example, it is good for simulating ground vehicles, legged creatures, and moving objects in virtual reality environments. It has built-in collision detection.

ODE is used for simulating the dynamic interactions between bodies in space. It is not tied to any particular graphics package. It supports several geometries: Box, sphere, capsule (cylinder capped with hemispheres), cylinder and other general geometrics.

ODE was started in 2001 and has already been used in many applications and games, such as BloodRayne 2, Call of Juarez, S.T.A.L.K.E.R and more importantly World of Goo the original game that we are carrying into 3D.

The game is highly dependent on physics simulation. The player builds complex structures of balls connected with elastic joints in the presence of gravity. So, it's quite not easy to build a robust physics engine to simulate the mentioned effects in short term period. Instead, we decided to use a ready one and to use it to create more complex effects like wind and water.

2.4 glFont

GLUT does not support wide manipulation of text. They support a very limited set of fonts. Therefore, we needed to find a utility by which we can display text easily in our application. We used glFont2 library for text manipulation.

glFont is a Win32 program that creates a texture containing a range of characters in a specified font, and automatically generates texture coordinates for use in OpenGL applications that need to display text. The font type is not limited to monospaced fonts; each character is displayed with correct spacing and size.

Text is rendered by texture mapping specific characters onto OpenGL quads. It is quite possible and easy to modify the color, size, and position of the text quads using standard OpenGL functions. It is also quite possible to load as many different fonts as you wish, and use them whenever you please.

Actually, the glFont library APIs were not sufficient to easily manipulate the text in our game. Therefore, we developed another set of methods on top of glFont to support displaying texts in many ways.

3. Implementation

3.1 Game design

The game transitions between a number of predefined states illustrated in the diagram below. We designed a simple mechanism for the transfer of control between game parts that handles the internal GLUT bindings and supports a modular code organization. This modular approach facilitated the introduction of more modules than those initially planned for.

The core module is called GlutController. It is an abstract class that defines the interface for other controllers and delivers glut events to the handlers provided by each controller. It also implements the transfer of control mechanism through two methods: receiveConrol and releaseControl. A controller can transfer control to the next controller by calling recieveControl on that controller before releaseing the control.

The version we delivered consists of three controllers:

3.1.1 IntroController

This controller receives control when the game starts up. It introduces the user to the game by driving the camera through a scene where animated goo balls build a random structure under wind and falling spheres. The controller then transfers control to the MainMenuController.

3.1.2 MainMenuController

When the intro is over this controller displays the game menu to the user. The user can start a new game, configure settings, learn about the game, check out the credits or exit the game. When a new game is started, control is transferred to the GameController.

The MainMenuController is also responsible for pausing the game. From inside the game, the user can pause the game by returning temporarily to the menu before resuming his game.

3.1.3 GameController

The game controller is the central piece of the game. It is responsible for rendering the game, applying physics, handling user inputs and updating the game. The GameController takes the user through a number of levels. At each level, the user will have to win under different constraints and level conditions.

The GameController relies on a number of methods to present the user with an interesting set of features to make the game playful and challenging.

3.2 Physics

Physics is the soul that moves and animates most of the game (if not all). It adds a variety and randomness to objects behaviors making those objects look very realistic.

ODE is procedural. It returns IDs to created physical entities then these IDs are passed to functions to manipulate the corresponding entities. Although this approach is very simple that makes ODE a very easy tool to be included in any program structure, there should be initial work to wrap ODE to be ready for usage in the game structure so as to provide extensibility and generality.

There are two main components in a typical ODE usage: world and space. A world is the entity that holds physical objects to apply physics laws on like the three Newtonian laws of movement. A world is responsible for changing position and orientation of objects. A space is the entity that performs collision detection.

ODE also provides bodies. A body is defined by a mass and a geometry. The mass defines the distribution of the total value of the mass and the distribution of the mass over the body (inertia) hence the mass is important for the physics simulator. The geometry defines the shape of the body hence it is helpful to check for collisions of that body with other ones.

There are also joints that defines constraints of movements between two bodies or between the body and the surrounding world.

A typical simulation will proceed like this:

1. Create a dynamics world.
2. Create bodies in the dynamics world.
3. Set the state (position etc) of all bodies.
4. Create joints in the dynamics world.
5. Attach the joints to the bodies.
6. Set the parameters of all joints.
7. Create a collision world and collision geometry objects, as necessary.
8. Create a joint group to hold the contact joints.
9. Loop:
    (a) Apply forces to the bodies as necessary.
    (b) Adjust the joint parameters as necessary.
    (c) Call collision detection.
    (d) Create a contact joint for every collision point, and put it in the contact joint group.
    (e) Take a simulation step.
    (f) Remove all joints in the contact joint group.
10. Destroy the dynamics and collision worlds.

From our side, we created the following entities:

GooWorld: It's a physics world that has gravity and holds all balls and joints created through the game playing time. It also has a sky and a ground.

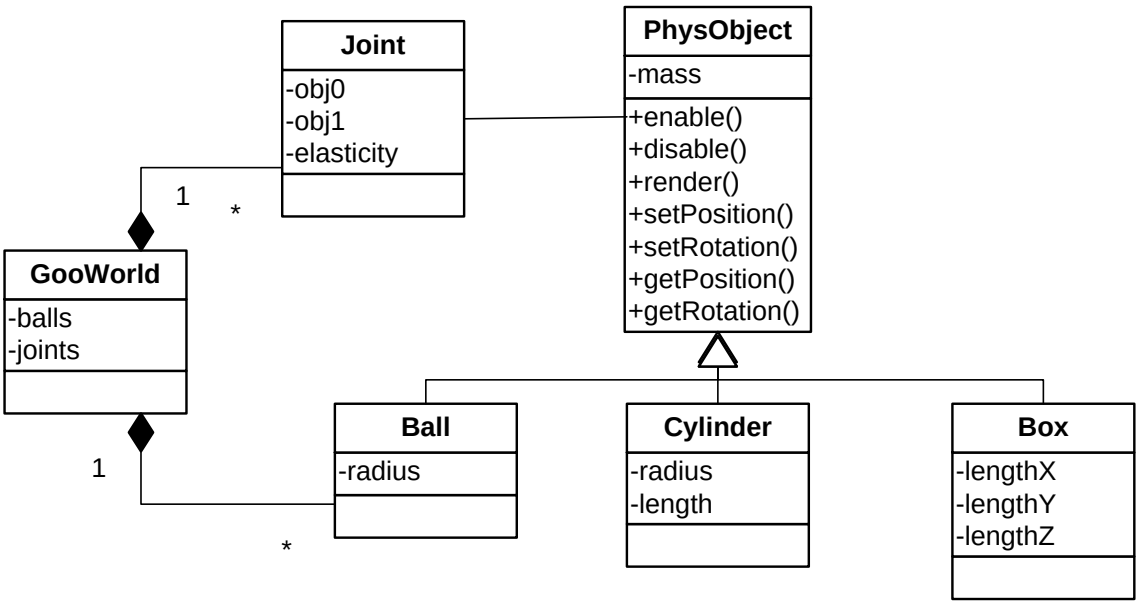
PhysObject: is an abstract physical object that has mass and could be rendered. All other physical objects extend this entity.

Ball: is a spherical physical object that represents the goo ball.

Joint: is an elastic constraint of the relative position of two different physical objects.

There are other physical objects like Cylinder and Box.

The following UML diagram shows the structure and relations among these entities.



3.3 User Interaction

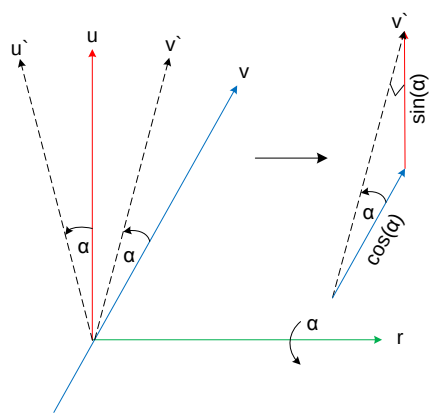
3.3.1 First Person Shooter Camera

We implemented a FPS camera that supports rotating and strafing in all directions. The mathematical model of the camera is based on the following three observations:

1. The three camera axes are orthogonal.
2. The cross product of any two axes results in the third one.
3. Rotations and strafing are axis-wise w.r.t the camera axes.

According to that, we needed a camera model that has its own conceptual axes, rather than using the world X, Y, and Z axes. The camera own axes are: right (r), up (u), and direction (v).

Using that model, camera rotation around the right axis is calculated as follows:



$$\vec{v`} = \vec{v} \cdot \cos(\propto) + \vec{u} \cdot \sin(\propto)$$
$$\vec{u`} = \vec{v`} \times \vec{r}$$

Rotations around the up, and direction axes are similarly performed.

As the camera now have its own right, up, and direction axes, strafing is implemented by adding a scalar step value, multiplied by the required direction, to the current camera location. For example, strafing to the left is represented by:

$$\vec{p`} = \vec{p} + \vec{r} \cdot (-step) \quad ; p \text{ is the current camera location}$$

After modifying the camera location or directions, the changes are applied to the 3D MODELVIEW matrix through the gluLookAt function as follows:

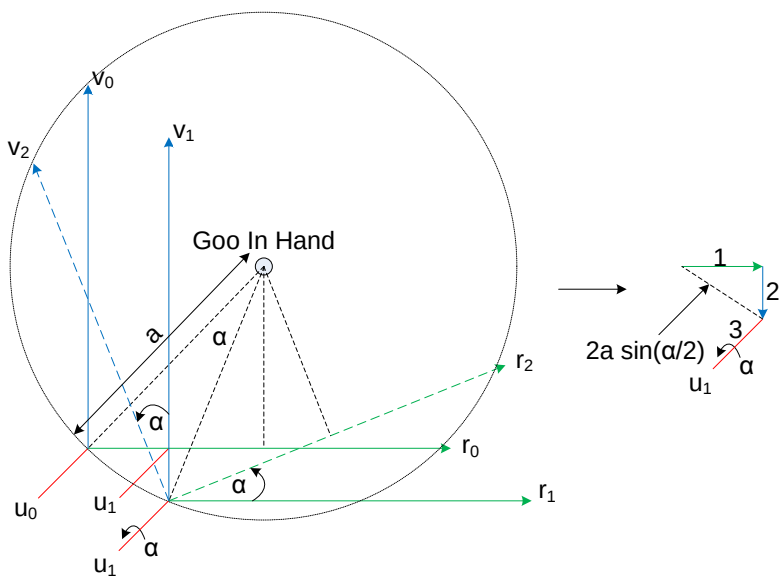
```
Vector3f center = location.add(direction);
gluLookAt(location.x, location.y, location.z,
          center.x, center.y, center.z,
          up.x, up.y, up.z);
```

3.3.2 Goo-Centric Rotation

In our game, the camera not only strafes and rotates around its own axes, but also it rotates around the new Goo ball (the Goo in hand). To rotate the camera around the Goo ball in the right direction, the following steps are performed:

1. Move the camera to the right.
2. Move the camera back (opposite to v direction).
3. Rotate the camera around the u axis to the left.

The following figure illustrates the applied steps:



Rotations in other direction are modeled the same way.

3.3.3 3D Object Selection

OpenGL has three rendering modes:

1. Render mode (`GL_RENDER`): used to draw on the frame buffer.
2. Select mode (`GL_SELECT`): used to select labeled objects.
3. Feedback mode (`GL_FEEDBACK`): used to produce lots of information about the scene being drawn.

The trick behind selection is that the `glRenderMode` function returns a value which doesn't depend on the new mode that we are passing, but on the previous rendering mode. The idea is that if we are in the `GL_SELECT` mode, and we used a counter that is incremented every time we draw something visible, then

we limited the drawing area to the very few pixels surrounding the mouse location, the result is that the counter is incremented only for the objects in that area.

Knowing only the number of objects in the mouse area is not enough to select one of them, though. To get that accomplished, we need the selection buffer. The selection buffer is a buffer managed by OpenGL that is filled with data about the object being drawn. Using that buffer, we can poll a list of objects in the cursor area.

OpenGL has no concept of objects. Instead, a name stack is used to pop and push names accordingly in the rendering mode. So that, before we draw any object, we push a name on the name stack, and when the drawing is done, we can push another name for the next object, and so on.

In our game, as we only want to move the current goo ball, we overwrite the peek element in the name stack with a `GOO_IN_HAND_ID`, namely an integer value, and draw the goo ball at the end of the rendering method so that the peek element remains synchronized with the goo ball.

The following code snippet implements the described selection technique:

```
GLuint buff[64];
GLint hits, view[4];
glSelectBuffer(64, buff);
glRenderMode(GL_SELECT);
glInitNames();
glPushName(0);
glMatrixMode(GL_PROJECTION);
glPushMatrix();
  glLoadIdentity();
  glGetIntegerv(GL_VIEWPORT, view);
  gluPickMatrix((GLdouble)x, (GLdouble)(window_height-y), 1.0, 1.0, view);
  gluPerspective(45.0, window_width/window_height, 0.0001, 1000.0);
  drawScreen();
  glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
hits = glRenderMode(GL_RENDER);
for (int i = 0; i < hits; i++) {
  int objId = (GLubyte)buff[i * 4 + 3];
  if (objId == GOO_IN_HAND_ID) {
    captured = true;
    break;
  }
}
```

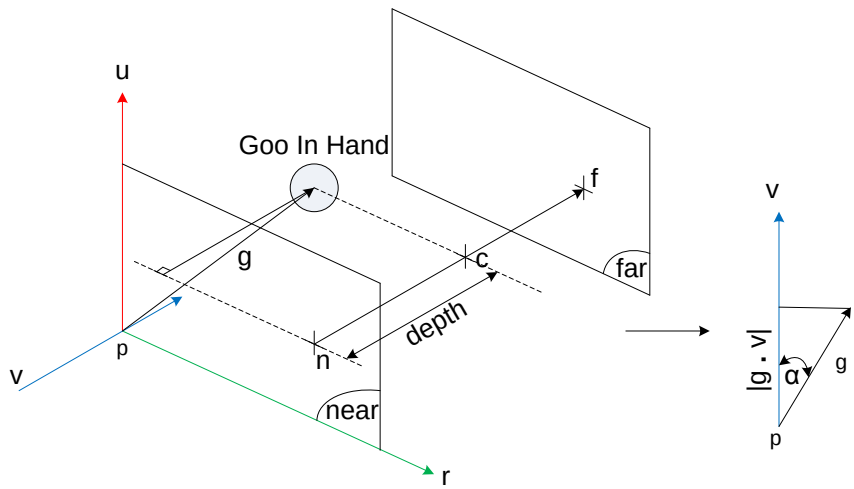
3.3.4 3D Picking Using Ray Casting

To drag the goo ball around the 3d world with the mouse, we needed a model to transform the 2d mouse location to 3d world coordinates, calculate the displacement value and direction, and apply the required transformations on the goo ball.

To transform the 2d mouse coordinates to world coordinates, we proceeded as follows:

1. Cast a ray from the near plane at z =0, to the far plane at z = 1;
2. Calculate the required depth, in the camera direction, where the goo ball is positioned. i.e. the distance between the goo ball and the <up,right> plane.
3. Add that depth to the near point to get the 3d mouse coordinates.

The following figure illustrates the explained model:



$$\vec{d} = \frac{f - n}{\|f - n\|}$$

$$\cos(\propto) = |\hat{g} \cdot \vec{v}| \,;\, \hat{g} = \frac{\vec{g}}{\|\vec{g}\|}$$

$$depth = \|\vec{g}\| \cdot \cos(\propto)$$

$$\vec{d} = \vec{d} \cdot depth$$

$$c = n + d$$

To calculate the displacement, we used two consecutive mouse locations, and applied the previous procedure on them to get two 3d mouse coordinates. Then, the Euclidean distance between the two 3d points is calculated, and the Goo ball is moved to the right and up directions of the camera using the cosine and sine values of the calculated displacement respectively.

The following snippet implements the 2d to 3d translation procedure:

```
GLint viewport[4];      glGetIntegerv(GL_VIEWPORT, viewport);
GLdouble projMat[16];   glGetDoublev(GL_PROJECTION_MATRIX, projMat);
GLdouble mvmatrix[16]; glGetDoublev(GL_MODELVIEW_MATRIX, mvmatrix);
GLdouble wx, wy, wz;
```

```
y = viewport[3] - y -1;
gluUnProject((GLdouble)x, (GLdouble)y, 0.0,
                          mvmatrix, projMat, viewport, &wx, &wy, &wz);
Vector3f n(wx, wy, wz);
gluUnProject((GLdouble)x, (GLdouble)y, 1.0,
                          mvmatrix, projMat, viewport, &wx, &wy, &wz);
Vector3f f(wx, wy, wz);
Vector3f d = f.add(n.mult(-1));
d = d.normalize();
Vector3f gooPos = prevGooPos;
Vector3f loc = camera.getLocation();
Vector3f dir = camera.getDirection();
Vector3f gl = gooPos.add(loc.mult(-1));
GLfloat glLength = sqrt(gl.x*gl.x + gl.y*gl.y + gl.z*gl.z);
GLfloat cosNormal = abs(gl.normalize().dot(dir));
GLfloat depth = glLength * cosNormal;
d = d.mult(depth);
Vector3f c = n.add(d);
return c;
```

3.3.5 Asynchronous Input Handling

GLUT keyboard handlers are synchronously executed. That's if a key is down, and the handler procedure is not finished yet, then another key is pressed, GLUT will terminate the first handler, and execute the second one. To support asynchronous key handling, all keys handling logic must be separated from keyboard handling callbacks, and the callback body must be as short as possible.

We used a key pool to implement asynchronous key handling as follows:

1. When a keyboard handler is called, it changes the corresponding key state in the keys pool.
2. In the game loop, and before the scene is updated, the required keys states are fetched from the pool, and the handling logic is applied accordingly.

3.4 2D Components

The game needed some 2D visual components on top of the 3D model. This was needed in menus, score bar and win/lose windows. After drawing the 3D objects, we draw the 2D things through the following procedure:

```
void drawScreen() {
   glutSetCursor(GLUT_CURSOR_NONE);
   glClearColor(0.0, 0.0, 0.0, 1.0);
   glClear(GL_COLOR_BUFFER_BIT);

   draw3DScene();
   draw2DScene();
   glFlush();
   glutSwapBuffers();
}

void draw2DScene() {

   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
   glLoadIdentity();
   glOrtho(0, screenWidth, 0, screenHeight, -1, 1);
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
   glLoadIdentity();

   /*    Draw 2D Objects   */

   glMatrixMode(GL_PROJECTION);
   glPopMatrix();
   glMatrixMode(GL_MODELVIEW);
   glPopMatrix();
}
```

3.5 Intelligent moving balls

One interesting aspect of the game is having non-attached balls (*free balls)* that wander through the structure. These balls also rush to the pipe when the user manages to get there by the end of each level. The structure is represented as a set of balls and a set of joints with each joint linking two balls together. New free balls are initialized to any of the available joint and given an initial direction on that joint. This behavior relies on a movement mechanism and two algorithms:

3.5.1 Moving the balls through the structure

It was going to be difficult to rely on the physics engine in maintaining the constraints of moving the balls along the joints. We decided to avoid that and do it ourselves. It was particularly interesting because the structure is constantly shaking. Almost all the structure balls change their position continuously and as a result the length of each joint is not constant with time.

We came to represent the movement of a ball by storing the ratio of the joint length the ball has cut so far. At each update, this ratio is incremented and the ball position is computed by moving that ratio of the distance away from the source ball towards the target ball.

3.5.2 Random graph traversal

The purpose of this algorithm is to enable the free balls to keep on traversing the structure. When a ball reaches one end of the joints it looks through the list of joints for all the joints attached to that ball excluding the one it just crossed. It then chooses one of these joints at random to continue traversing.

3.5.3 Pathfinding

When a ball comes close enough to the pipe, the user wins the level. We compute dijkstra's shortest paths from each node in the structure and that node. The free moving balls then choose the joint that belongs to the shortest path tree instead of choosing a random joint.

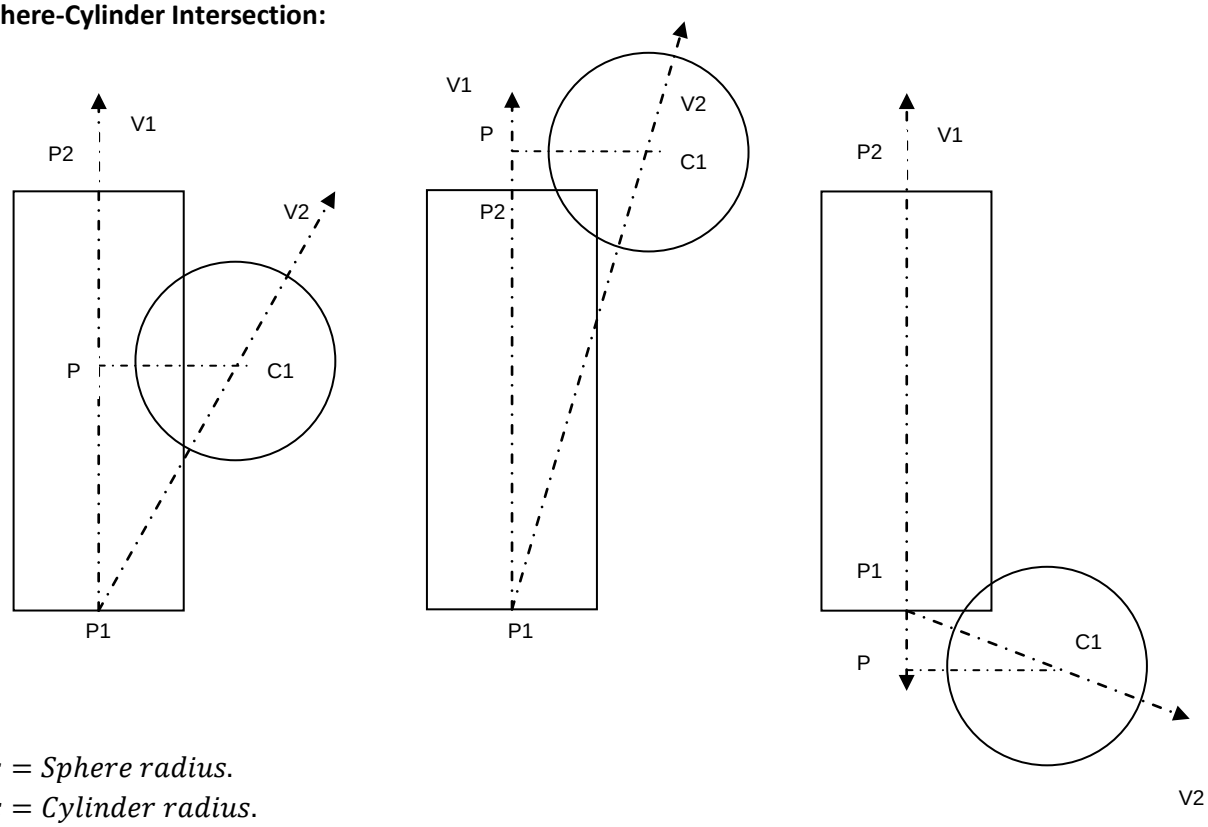
3.6 Collision Detection

In our game, we assume that a ball can't intersect with another ball or with a joint. We developed two methods for checking if an intersection occurred:

Sphere-Sphere Intersection:

Two spheres intersect if the distance between their centers is less than the sum of their radii.

Sphere-Cylinder Intersection:



$Rs = Sphere\ radius.$
$Rc = Cylinder\ radius.$
$L = Cylinder\ height.$
$V1 = P2 - P1$
$V2 = C - P1$

A sphere intersects with a cylinder *iff* these two conditions hold:

- $V1 \cdot V2\ is\ in\ range\ [-Rs, L + Rs]$
- $\|CP\| \leq Rs + Rc$

Note that these two conditions may hold in a case in which there is no intersection, but in that case, the cylinder and the sphere will be very close so that we can consider them intersecting. We believe we can ignore this case as the game can tolerate that.

4. Features

4.1 Physics

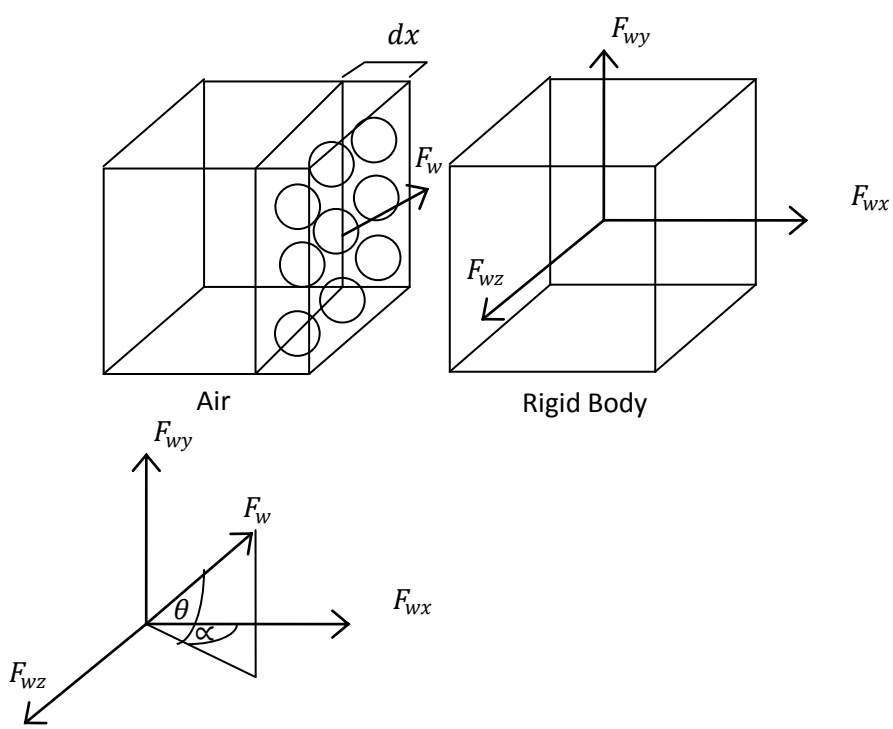
We built up two features on top of the ODE physics engine: the wind and water effects.

4.1.1 Wind

Our wind model is based on the following assumptions:

1. Air consists of identical particles of constant radius r_p and constant mass m_p.
2. All air particles at a time instant move with a constant acceleration in the same direction $\overrightarrow{a_p}$.
3. Air particles are uniformly distributed through any volume with constant density of ρ.
4. Distance between each two particles is far enough to neglect the effect of each particle on another.

From the previous assumptions it's valid to say that distances among all neighbor particles are equal also they are of constant value as long as density and acceleration are not changed. We can also say that there is a kind of similarity so that there are cases where the effect of n particles is equivalent to the effect of one particle times n.



Let's say that there is a slice of air affects one surface of a rigid box as shown in the previous figure. We can write:

$$\overrightarrow{F_w} = M\overrightarrow{a_p}$$

$$M = \rho V$$

$$V = w.h.dx$$

$$dx = 2r_p$$

$$A = w.h$$

$$\therefore M = 2\rho A r_p$$

$$\therefore \overrightarrow{F_w} = 2\rho r_p A\overrightarrow{a_p}$$

For any rigid body we can assume that A is the surface area and $\overrightarrow{F_w}$ is the wind force that affects that surface.

So, our wind is defined by the following parameters:

ρ: Air density

r_p: Air particle radius

$\overrightarrow{a_p}$: Vector (direction and magnitude) of the acceleration of air particles

A: Surface area of the rigid body exposed to the wind

Before each simulation step the wind force is calculated for each body in the simulation and added to its force accumulator.

```
class Wind
{
public:
      Wind(float airDensity
                  , float airParticleRadius
                  , float windAcceleration
                  , float windDirectionX
                  , float windDirectionY
                  , float windDirectionZ);
      ~Wind(void);
      void affectPhysObject(PhysObject *physObject, float surfaceArea);

public:
      float airDensity;
      float airParticleRadius;
      float windAcceleration;
      float windDirectionX;
      float windDirectionY;
      float windDirectionZ;
};
```

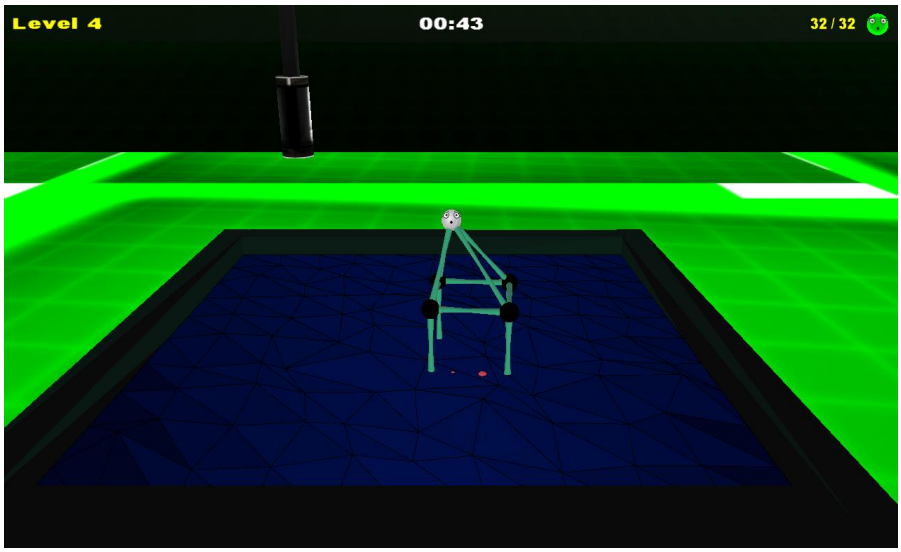
4.1.2 Water

Our water is represented by a grid of balls connected with elastic joints. The stronger the joints are the more viscous the liquid is. Balls represent the liquid particles and joints represent binds and attraction between particles. In order to cause disturbance to the surface of the liquid random forces with random magnitudes and directions are applied on all balls of the grid before each simulation step.

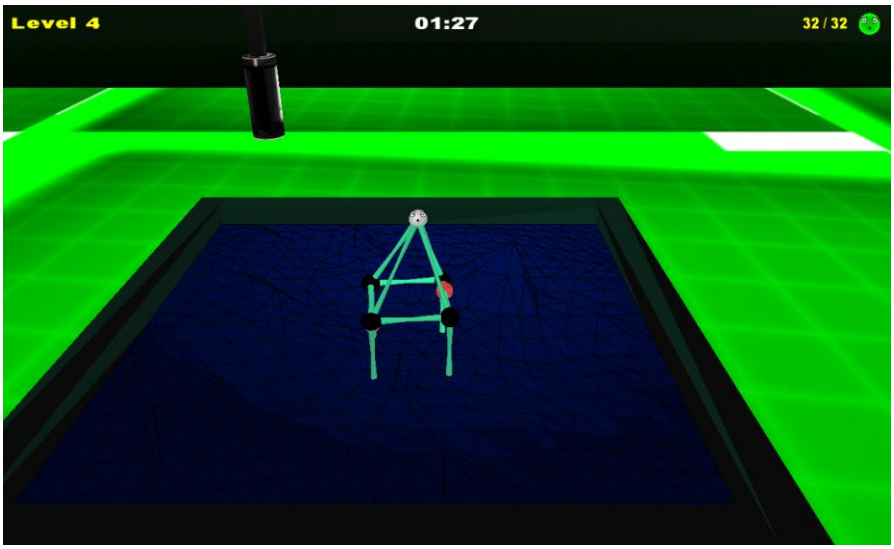
We thought of many approaches to render the water. It could be rendered as quads, triangles, Bezier surface or any other smoothed curvy surface.

We tried all approaches but some worked well and some didn't. We finally implemented the triangles approach due to scarce in time.

The following image shows water rendered as triangles with vertices corresponding to balls positions.



The following image shows water rendered as the previous triangles but subdivided into 4 triangles each which adds more variation to the surface:



```
for(unsigned i = 0; i < ROWS-1; i++)
      for(unsigned j = 0; j < COLS-1; j++) {
            subdiv(balls[i][j]->getPosition(), balls[i][j+1]->getPosition(),
balls[i+1][j+1]->getPosition(), 0);
            subdiv(balls[i][j]->getPosition(), balls[i+1][j+1]-
>getPosition(), balls[i+1][j]->getPosition(), 0);
      }



void subdiv(const dReal v0[3], const dReal v1[3], const dReal v2[3], int
depth)
{
      if(depth == 0) {

            glColor4f(0, .1, .7, .5);
            glBegin(GL_TRIANGLES);
            glVertex3fv(v0);
            glVertex3fv(v1);
            glVertex3fv(v2);
            glEnd();

            //glColor4f(0, 0, 0, .5);
            //glBegin(GL_LINE_LOOP);
            //glVertex3fv(v0);
            //glVertex3fv(v1);
            //glVertex3fv(v2);
            //glEnd();


      } else {

            dReal v01[3];
            dReal v12[3];
            dReal v20[3];

            for(int i = 0; i < 3; i++) {
                  v01[i] = (v0[i] + v1[i]) / 2.0;
                  v12[i] = (v1[i] + v2[i]) / 2.0;
                  v20[i] = (v2[i] + v0[i]) / 2.0;
            }

            int newdepth = depth - 1;
            subdiv(v0, v01, v20, newdepth);
            subdiv(v01, v1, v12, newdepth);
            subdiv(v01, v12, v20, newdepth);
            subdiv(v20, v12, v2, newdepth);

      }
}
```

subdiv:subdivides each triangle recursively into 4 triangles.

4.2 User Interaction

The user can move the Goo ball around the world using both mouse and keyboard. Dragging a Goo ball will move it in the 2D space facing the camera, and pressing the Q and Z keyboard keys will move the Goo ball, in and out respectively, in the direction of the camera.

The Goo ball can also be moved using the A, S, W, D keys.

To control the camera, the user can use the arrow keys to rotate in all directions, in addition to the PAGE DOWN, PAGE UP, HOME, and END keys to move the camera in, out, up, and down respectively.

The camera also follows the mouse when the mouse reaches the bounds of the screen.

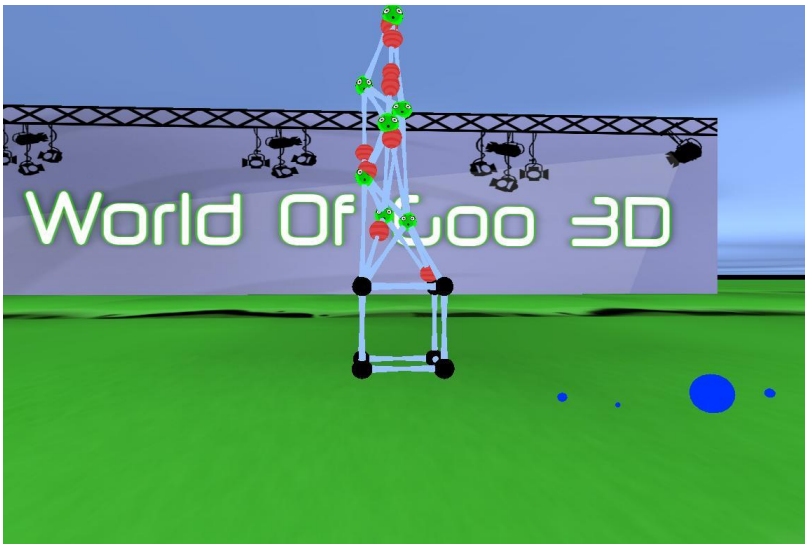
4.3 Game Intro

The intro illustrates the game in general. There are three stages in the intro.

The first stage starts with an empty green scene where you can see the clouds are moving rapidly because of winds. The camera moves quickly towards some sort of structure. As the camera gets closer, you can see the "World of Goo" written in the background. Finally, the camera stops at a structure made of Goo balls.

The second stage now starts. The camera starts rotating around the structure. During this, the gray Goo ball moves randomly building up the Goo structure. Furthermore, balls fall from everywhere in the sky giving a good effect.

After the camera rotates 2 times, the third stage starts when the camera faces the "World of Goo" background. The camera moves rapidly towards the background until it hits it. The main menu then appears.

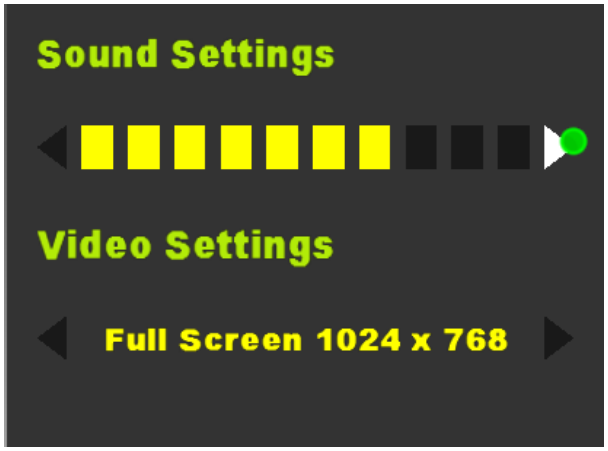


4.4 Interactive Menu

For high usability, we provide the user with an interactive menu that has visual and sound effects. Through the menu, the user is able to change the game settings, view help, view credits, start/resume game and exit.

4.5 Game Settings

The user is capable of changing the audio and video settings, even in the middle of a game. In the game settings window, we provide an interface to change the volume and the resolution of the screen.



4.6 Score Bar

While playing, the user is able to view the time and the number of remaining balls through a 2D transparent score bar that is displayed on the top of the screen. The bar contains the level number, time elapsed and number of remaining balls.

4.7 Pause/Resume

The user has the ability to pause the game and resume it again through the menu. This capability is provided by maintaining the state of all variables when going to the menu.

4.8 Lighting

There are four lights in the scene. The lights are defined as follows:

```
GLfloat diffuseLight0[] = {0.8f, 0.0f, 0.0f, 1.0f};
GLfloat specularLight0[] = {1.0f, 0.0f, 0.0f, 1.0f};
GLfloat ambientLight0[] = {0.8f, 0.0f, 0.0f, 1.0f};
GLfloat lightPos0[] = {0, light_y, -2, 0.0f};
GLfloat spotlightDirection0[] = { 0.0, 0.0, 1 };
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight0);
glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight0);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight0);
glLightfv(GL_LIGHT0, GL_POSITION, lightPos0);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 30.0f);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spotlightDirection0);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 10.0f);
```

The rest of the lights are defined in a similar manner. The spot exponent indicates the concentration of the light towards the direction of the light, while the cut off angle indicates the angle of spread of the light.

The lights are divided into 2 types:

1. Moving lights: The first two lights are in continuous opposite movement along the y-axis. They update their location every 1 second.
2. Stationary lights: The last two lights are stationary and are on both sides of the structure. These lights are turned on and off randomly every 5 seconds. They act like spot lights on the structure. That gives a good visual effect on the structure.

4.9 Textures

All the texture images are taken from the 2D game. Each level has its own set of textures: background, ground, goo balls. Textures were applied on three kinds of shapes:

1. **Quad**
   The following is an example of quad textures (the snippet is used to place a texture on the background):

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, bgTexture[currentLevel]->id);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glBegin(GL_QUADS);
   glNormal3f(0.0, 0.0f, 1.0f);
   glTexCoord2f(0.0f, 0.0f); glVertex3f(bg_left, bg_bottom, bg_far);
   glTexCoord2f(1.0f, 0.0f); glVertex3f(bg_right, bg_bottom, bg_far);
   glTexCoord2f(1.0f, 1.0f); glVertex3f(bg_right, bg_top, bg_far);
   glTexCoord2f(0.0f, 1.0f); glVertex3f(bg_left, bg_top, bg_far);
glEnd();
glDisable(GL_TEXTURE_2D);
```

2. **Sphere**
   The following was used to place textures on the GlutSpheres. Texture coordinates generation was enabled instead of calling glTexCoord.

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glBindTexture(GL_TEXTURE_2D, texture->id);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

3. **Cylinder**
   The cylinder is defined as a quadric object. We call gluQuadricTexture on the quadric object to allow the generation of texture coordinates when rendering a gluCylinder as follows:

```
GLUquadricObj* quadric = gluNewQuadric();
gluQuadricDrawStyle(quadric, GLU_FILL);
gluQuadricNormals(quadric, GLU_SMOOTH);
gluQuadricOrientation(quadric, GLU_OUTSIDE);
gluQuadricTexture(quadric, GL_TRUE);
```

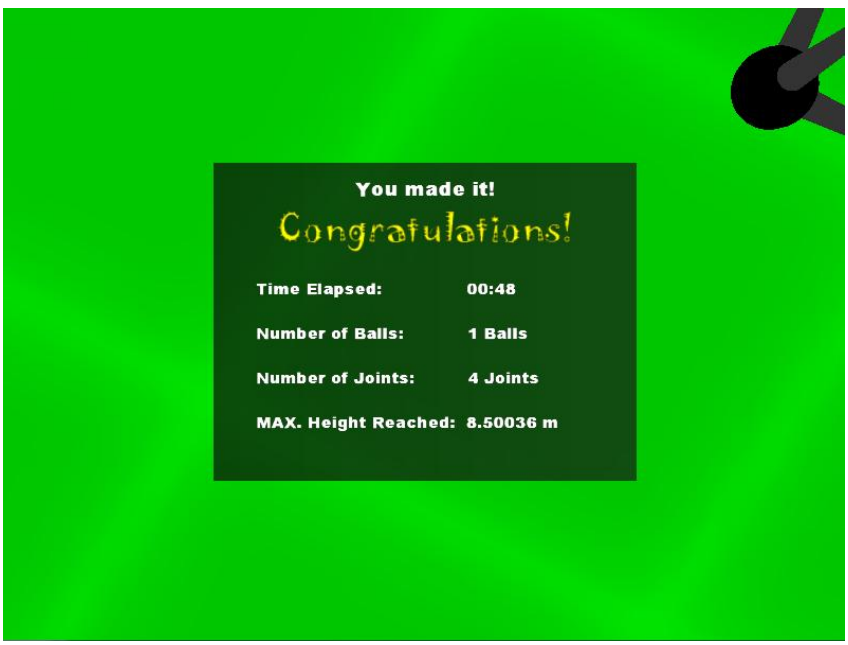
```
glEnable(GL_TEXTURE_2D);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glBindTexture(GL_TEXTURE_2D, pipeTexture->id);
```

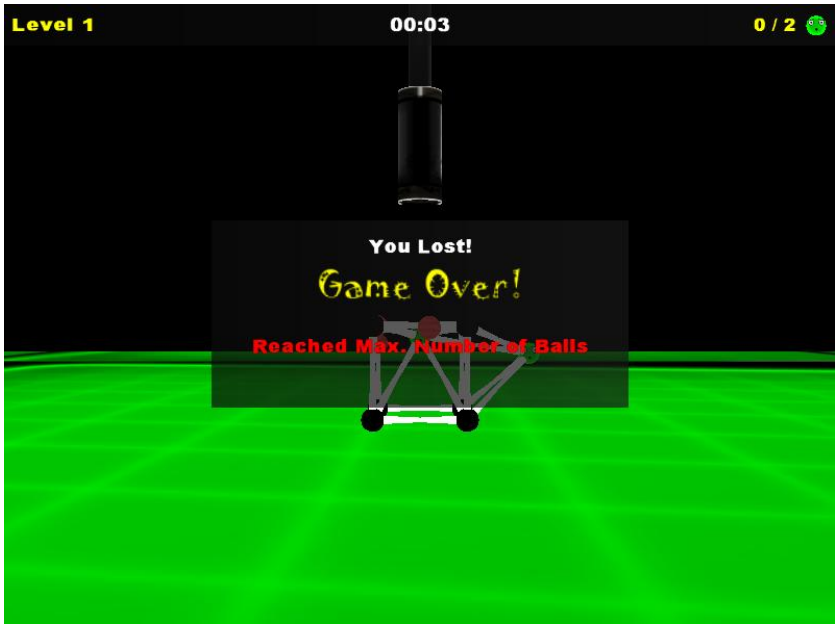
4.10 Random Sounds

Random sounds are played every 5 seconds. In addition, sounds are played when a ball is placed or when a player wins/ loses.

4.11 Win/Lose Effects

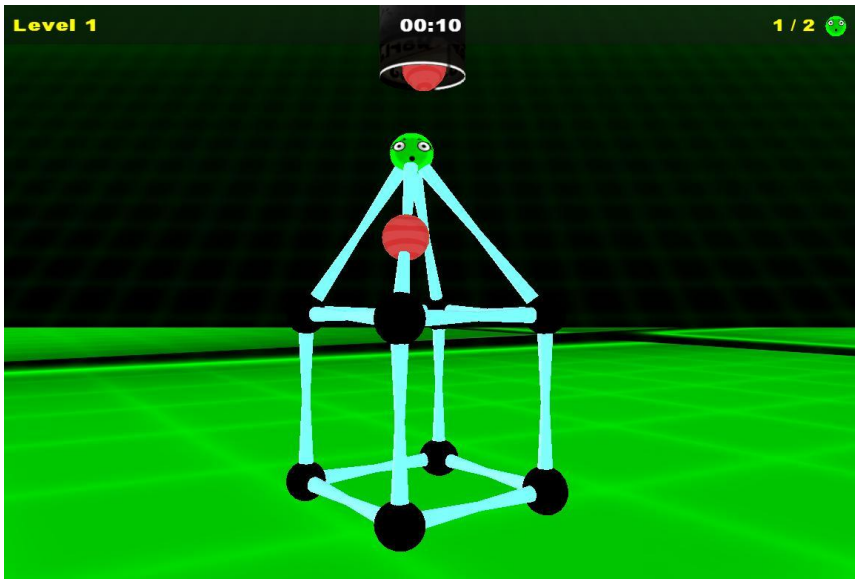
When the user wins/loses the game, a transparent window is displayed containing his state. If the user wins only, the camera is rotated continuously over the scene. In addition, there are sound effects beside the visual effects.





4.12 Moving balls

These balls wander through the goo structure; shaking joints and finally jumping through air to the pipe adding a lot of fun to the gaming experience.



5. Game Play

The game starts with an intro that shows an approaching background. Once the background becomes close enough a Goo structure appears. The camera then starts rotation around this structure. At the same time, the structure is built up randomly while balls start falling from the sky. After two rotations the camera goes towards the background and the menu then appears.

Each level starts with an initial structure, a pipe and a number of moving Goo balls. The player moves the Goo ball at hand to any position he desires. When the Goo ball at hand approaches another fixed Goo ball, a transparent joint appears. Before a Goo ball is placed, some collision detection checks are made to ensure that the new Goo ball doesn't overlap with any other Goo balls or joint. A Goo ball can only be added to the structure if it has at least three neighbors. The player can also move the camera freely with the set of controls provided. For every 5 Goo balls added to the structure, a moving Goo ball is created which increases the level of difficulty as the game continues.

When a Goo ball becomes close enough to the pipe the player wins and the moving Goo balls move to the pipe. The game then advances to the next level. There is a maximum number of balls that can be added, after that the player loses.