COMP371 - TEAM 17 REPORT ON SOFTBODY

By Jiarui Li 40167860

Zixuan Tang 40024456

Lingjue Zou 40106768

Yuhan Chen 40101970

Fuqiang Zhai 40072577

A report submitted in partial fulfillment of the requirements of COMP 371

Concordia University

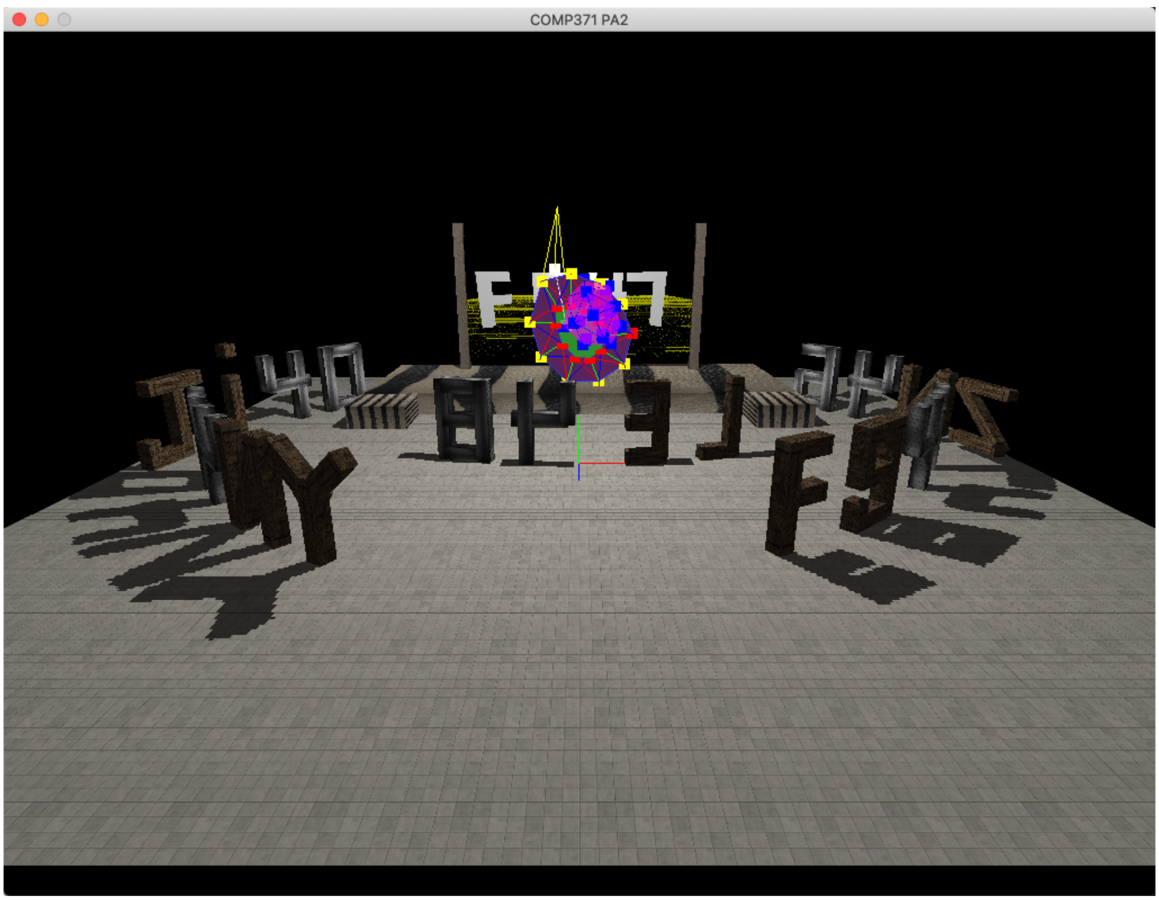
February 2021

Contents

1. INTRODUCTION AND OBJECTIVE

The topic of our team is SOFTBODY Simulation. The goal of this project is to introduce three SOFTBODY objects into our PA2 scenario, displaying three objects on the stage. Users can interact with the SOFTBODY in the scene by dragging the mouse. We also retain the texture mapping, lighting, shadowing, model transformation, and a series of other functions from PA2.

The following is our final project demonstration diagram:



To realize this project, we divided the whole project into three parts:

The first part is to modernize the source code of SOFTBODY Simulation because the SOFTBODY Simulation source code provided by OpenISS Framework is developed based on OpenGL 2 and GLUT framework. We need to replace all its rendering methods related to OGL 2 with OGL 3 rendering methods and use GLFW and GLEW to create windows so that SOFTBODY can successfully render entities in PA2 based on OpenGL 3.

The second part is to forge the modern SOFTBODY Simulation into a library-like project. The purpose of this is for other people can directly reference source code residing in the cloud or local library without having to copy many source files every time. In addition, we added several designs to allow referees to modify SOFTBODY source files minimally to achieve their desired goals (at least we have added enough features to avoid the complexity of modifying the SOFTBODY source code when introducing into PA2).

The last part is to import the previously prepared SOFTBODY into our PA2 scene and let them show on the stage. We need to create the corresponding 1D, 2D, and 3D objects in main function and render them in real-time. Also, the human-computer interaction of the mouse needs to be added.

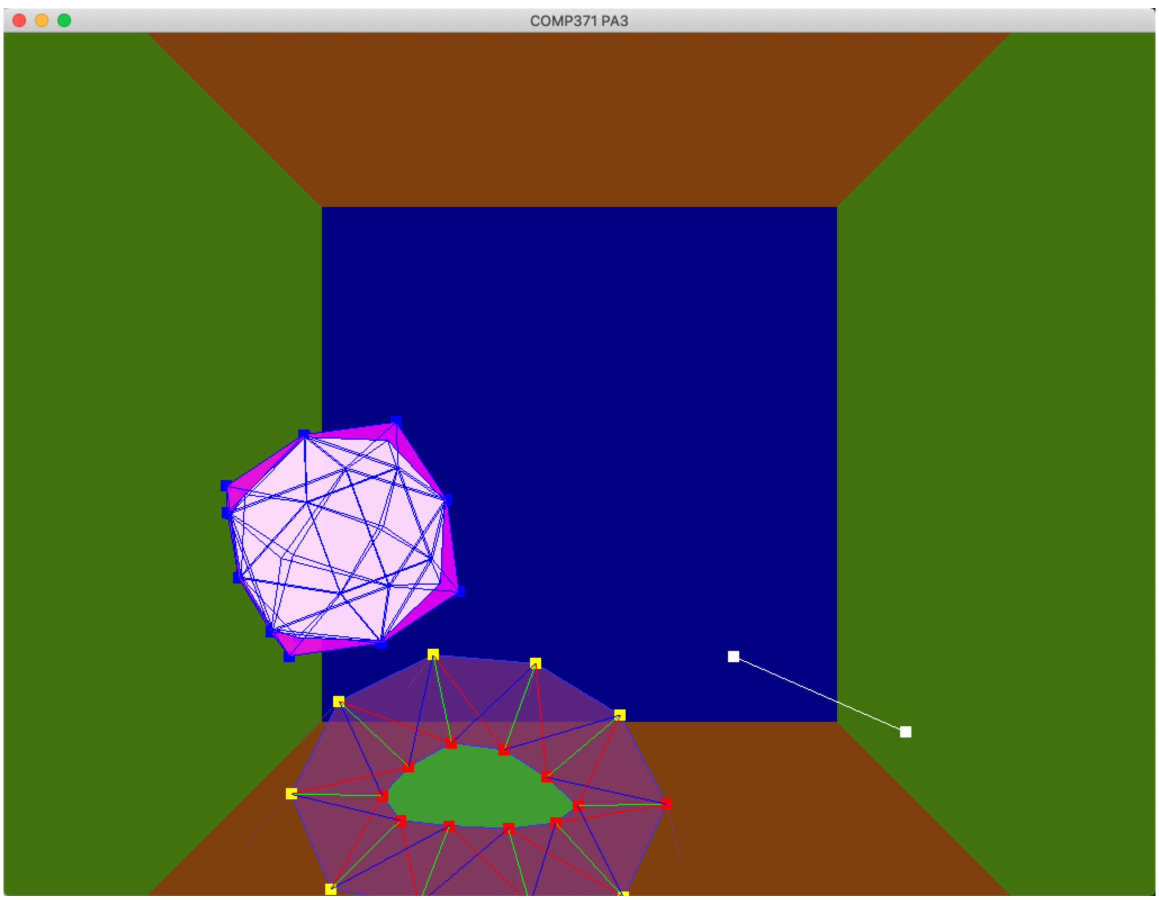
2. DEVELOPMENT AND TECHNIQUES

2.1 Modernizing SOFTBODY Simulation

2.1.1 Background

As mentioned in the previous section, the SOFTBODY source code provided in the OpenISS framework is developed based on OGL 2 and GLUT. We need to modernize the source code to use OGL 3, GLFW, and GLEW to render SOFTBODY objects. We have divided the modernization work into two parts: the first part is to modernize the drawing function; the second part is to modernize the main function for rendering. The following two subsections will discuss the specific methods and logic we used.

The following is a diagram showing the results of the successful modernization.



2.1.2 The Drawing function using Dynamic Buffer

For SOFTBODY, most of the code, such as Integrator, Spring, Vector, Particle, and Object excluding drawing functions, are mathematical logic of SOFTBODY simulation itself. They do not involve the use of OpenGL, which means we only need to detach the *Draw()* function of Object from OpenGL 2 and apply OpenGL 3 to it.

Now let's briefly talk about the difference between OpenGL2 and OpenGL3 rendering code. In OGL2, for a geometric figure to be rendered, the following code is used:

```
glPushMatrix();
    glColor4f(0.0,1.0,0.0, 0.5 );
    glBegin(GL_POLYGON); // the draw of inner circle
    for(i=0 ; i<inner_points.size(); i++)
    {
        glVertex2f(inner_points[i]->r->x, inner_points[i]->r->y);
    }
    glEnd();
glPopMatrix();
```

(code reference： https://github.com/OpenISS/SoftbodySimulationSystem)

Use *glColor4f ()* to set the color and alpha value of the fragment, and then pass in the render mode as parameter to *glBegin()* which starts the drawing process, ending at *glEnd ()*. Between the begin and end functions, *glVertex2f ()* or *glVertex3f ()* are all geometric vertices to be drawn.

However, in OGL3, rendering is done by vertex array, buffer, and shader. First, we need to create a vertex array for the vertices to be drawn, create a buffer for the array, and upload the buffer to the GPU through *glBindVertexArray()* and *glBindBuffer()*. Then call *glDrawArrays(),* specify the render mode and the vertex offset and vertex count to be drawn as parameters of this function. Finally, the vertex shader and fragment shader cooperate to complete the drawing of the graphics. Despite the complicated operation, it also gives us more freedom when drawing. For example, we can give vertex array different attributes (such as vertex colors, vertex normal, vertex UV, etc.) though *glVertexAttribPointer()*. Then, cooperate with the programmable shader to change the color of our graphics or add textures, lights, shadows, is more efficient.

Backing to the discussion of modern SOFTBODY, each SOFTBODY object has data fields as shown in the figure below to specify their shape and position.

```
//##ModelId=45F4D79701E6
class Object
{
protected:
    //##ModelId=45F4D79701E7
    int numParticles;                          // number of particles of the object
    //##ModelId=45F4D79701F5
    int numSprings;                            // number of springs of the object
    //##ModelId=45F4D79701F7
    Integrator* integrator;                    // integrator object
    //##ModelId=45F4D7970206
    integrator_type integratorType;            // different type of integrator, Euler, Midpoint, RK4

public:
    //##ModelId=45F4D797020B
    vector<Spring*>  inner_springs;            // the springs on the inner layer of the object
    //##ModelId=45F4D7970216
    vector<Spring*>  outer_springs;            // the springs on the outer layer of the object
    //##ModelId=45F4D7970227
    vector<Particle*>  inner_points;           // the particles on the inner layer of the object
    //##ModelId=45F4D7970235
    vector<Particle*>  outer_points;           // the particles on the outer layer of the object
    //##ModelId=45F4D7970246
    vector<Face*> inner_faces;                 // the surfaces on the inner layer of the object
    //##ModelId=45F4D797024B
    vector<Face*> outer_faces;                 // the surfaces on the outer layer of the object
    //##ModelId=45F4D7970254
    dimensionality dim;                        // dimension type to specify the object is 1D, 2D, or 3D

    //##ModelId=45F4D7970258
    int   closest_i ;                          // the closest point on the outer layer to mouse position

public:
    //##ModelId=45F4D7970263
```

(code reference: https://github.com/OpenISS/SoftbodySimulationSystem)

These data are stored in C++ STL container, and every time the *Update()* function of Object is called, the data will be updated according to the current state and external forces. The figure below is *Update()*; the function call in the red box is the integrator algorithm that updates the object's data.

```cpp
//##ModelId=45F4D7970274
void Object::Update(float deltaT, bool drag, float xDrag, float yDrag)
{

    if(integrator == NULL)
    {

        switch(integratorType)
        {
            case EULER:
                integrator = new EulerIntegrator(*this);
                break;

            case MIDPOINT:
                integrator = new MidpointIntegrator(*this);
                break;

            case RK4:
                integrator = new RungeKutta4Integrator(*this);
                break;

            default:
                assert(false);
                return;
        }

        integrator->setDimension(dim);
    }

    integrator->integrate(deltaT, drag, xDrag, yDrag);
}
```

（reference: https://github.com/OpenISS/SoftbodySimulationSystem）

Each time the data is updated, it needs to be drawn once to reflect the latest shape and position of the object on the screen. Below we will explain how to obtain the object data, create the corresponding vertex array through OpenGL3, and draw the related graphics. An object is composed of many sub-instances. We are taking *object2D* as an example to draw a complete *object2D*. We need to draw its outer normal, outer polygon, inner polygon, outer springs, inner springs, radium springs, shear springs, and the line between the integrator drag and the closest point on the object separately. Our design is to put all the vertices of these sub-instances into an array called *vertexArrayData*, using a pointer to maintain the access to the array. Then call the *updateVertexArray()* function we designed before each drawing to update the content in *vertexArrayData*. The following figure indicates what has been added to *object2D.h*.

```
// for OGL3 drawing
void createVertexArray();
glm::vec3* vertexArrayData;
void updateVertexArray();
unsigned int total;
GLuint VAO;
GLuint VBO;

// other helper functions to get the offset of drawing instances
enum drawing_instances{outer_line, outer_polygon, inner_polygon, outer_spring, inner_spring, radium_spring,
    shear_left, shear_right, integrator_line};

int getOffset(drawing_instances instance);
int getCount(drawing_instances instance);
```

In the constructor of *Object2D*, after *SetObject()* has been called, we call *createVertexArray()* to dynamically allocate memory for the *vertexArrayData* array. The amount of memory to be allocated is figured out by counting how many vertices we have to draw in total. Then we generate and bind the vertex array and vertex data, upload the empty (empty means it has not been initialized with values, but still occupies memory) *vertexArrayData* to buffer for the first time, and set the vertex attribute pointer correctly. The following figure is the function body of *createVertexArray().* Another critical point here is, we have to specify *GL_DYNAMIC_DRAW* mode when uploading buffer data, rather than *GL_STATIC_DRAW*. Because the data in *vertexArrayData* is updated before each rendering. As a result, we must use dynamic drawing to ensure that OpenGL can use dynamic buffer to track all updates.

```cpp
void Object2D::createVertexArray(){
    total = 0;
    // here's how the number of vertercies are counted
    total += outer_faces.size() * 2;        // for drawing outer lines
    total += outer_points.size();           // for drawing outer polygon
    total += inner_points.size();           // for drawing inner polygon
    total += outer_springs.size() * 2;      // for drawing outer springs
    total += inner_springs.size() * 2;      // for drawing inner springs
    total += radium_springs.size() * 2;     // for drawing radium springs
    total += shear_springs_left.size() * 2; // for drawing shear_springs_left
    total += shear_springs_right.size() * 2;// for drawing shear_springs_right
    total += 2;                             // for drawing the integrator line between mouse and object

    // dynamically allocate the vertexArrayData size by counting how many vertercies in total
    vertexArrayData = new glm::vec3[total];

    // generate VAO and VBO, upload the data, reserve memory in buffer
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);

    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(*vertexArrayData) * total, (void*)vertexArrayData, GL_DYNAMIC_DRAW);

    // set attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (void*)0);
    glEnableVertexAttribArray(0);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}
```

Next is the *updateVertexArray()* function. In this function, we first bind the buffer that needs to be updated through calling *glBindBuffer()* and then use a list of loops to retrieve all the vertices we need to draw. Then push back the vertices to a temporary vector called *tempVertices*.

```cpp
void Object2D::updateVertexArray(){
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // use a temp vector to update all the information and then copy the vector to vertexArrayData
    vector<glm::vec3> tempVertices;
    // push in the outer circle lines data
    for(int i = 0; i < outer_faces.size(); ++i){
        tempVertices.push_back(glm::vec3(outer_faces[i]->fp1->r->x, outer_faces[i]->fp1->r->y, outer_faces[i]->fp1->r->z));
        tempVertices.push_back(glm::vec3((outer_faces[i]->normal->x + outer_faces[i]->fp1->r->x),
                                         (outer_faces[i]->normal->y + outer_faces[i]->fp1->r->y),
                                         (outer_faces[i]->normal->z + outer_faces[i]->fp1->r->z)));
    }

    // push in the outer polygon data
    for(int i = 0; i < outer_points.size(); i++)
    {
        tempVertices.push_back(glm::vec3(outer_points[i]->r->x, outer_points[i]->r->y, 0.0f));
    }

    // push in inner polygon data
    for(int i = 0; i < inner_points.size(); ++i)
    {
        tempVertices.push_back(glm::vec3(inner_points[i]->r->x, inner_points[i]->r->y, 0.0f));
    }

    // push in outer spring data
    for(int i = 0; i < outer_springs.size(); ++i){
        tempVertices.push_back(glm::vec3(outer_springs[i]->sp1->r->x,outer_springs[i]->sp1->r->y, 0.0f));
        tempVertices.push_back(glm::vec3(outer_springs[i]->sp2->r->x,outer_springs[i]->sp2->r->y, 0.0f));
    }

    // push in inner spring data
    for(int i = 0; i < inner_springs.size(); ++i){
        tempVertices.push_back(glm::vec3(inner_springs[i]->sp1->r->x,inner_springs[i]->sp1->r->y, 0.0f));
        tempVertices.push_back(glm::vec3(inner_springs[i]->sp2->r->x,inner_springs[i]->sp2->r->y, 0.0f));
    }

    // push in radium spring data
    for(int i = 0; i < radium_springs.size(); ++i){
        tempVertices.push_back(glm::vec3(radium_springs[i]->sp1->r->x,radium_springs[i]->sp1->r->y, 0.0f));
        tempVertices.push_back(glm::vec3(radium_springs[i]->sp2->r->x,radium_springs[i]->sp2->r->y, 0.0f));
    }

    // push in shear spring left data
    for(int i = 0; i < shear_springs_left.size(); ++i){
        tempVertices.push_back(glm::vec3(shear_springs_left[i]->sp1->r->x,shear_springs_left[i]->sp1->r->y, 0.0f));
```

It is worth noting that when pushing integrator line vertex data in, we have to make sure the object's integrator is not null; otherwise, there'll be *BADACCESS* runtime error.

```cpp
// push in integrator line data
if(integrator != nullptr){
    tempVertices.push_back(glm::vec3(integrator->mDragX, integrator->mDragY, 0.0f));
    tempVertices.push_back(glm::vec3(outer_points[closest_i]->r->x, outer_points[closest_i]->r->y, 0.0f));
}
```

After all the loops are over, the length of *tempVertices* should match the length of the *vertexArrayData* we created earlier, or it is two vertices off the total length since integrator might be null at first. At this time, we copy all the data from *tempVertices* into *vertexArrayData*. To complete the update, we also need to upload the newly updated *vertexArrayData* to the GPU buffer corresponding to this object through *glBufferSubData()* (as shown in the figure below). *glBufferSubData()* is used in connection with the *GL_DYNAMIC_DRAW* mode as mentioned above. Thus, we have completed the vertex update before drawing.

```
    // copy the vector to vertexArrayData
    for(unsigned int i = 0; i < total; ++i)
    {
        // this if statement is to make sure the vertexArrayData's size is unchanged and every vertex is initialized
        // (in case integrator is null)
        if(i < tempVertices.size())
            vertexArrayData[i] = tempVertices.at(i);
        else
            vertexArrayData[i] = glm::vec3(0);
    }

    // update the buffer data
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(*vertexArrayData) * total, (void*)vertexArrayData);
}
```

Before using *glDrawArrays()* to render within the *Draw()* function, we need to know the offset and count of the vertex of each sub-instance from the *vertexArrayData* in the object. Therefore we added two functions, *getOffset()* and *getCount(),* taking a *drawing_instances* enum type defined in the header file as a parameter, then get the offset and count of each instance by the size of the vector in the object.

```
int Object2D::getOffset(drawing_instances instance){
    int offset = 0;

    switch(instance){
        case outer_line:
            offset = 0;
            break;
        case outer_polygon:
            offset = (int) outer_faces.size() * 2;
            break;
        case inner_polygon:
            offset = getOffset(outer_polygon) + getCount(outer_polygon);
            break;
        case outer_spring:
            offset = getOffset(inner_polygon) + getCount(inner_polygon);
            break;
        case inner_spring:
            offset = getOffset(outer_spring) + getCount(outer_spring);
            break;
        case radium_spring:
            offset = getOffset(inner_spring) + getCount(inner_spring);
            break;
        case shear_left:
            offset = getOffset(radium_spring) + getCount(radium_spring);
            break;
        case shear_right:
            offset = getOffset(shear_left) + getCount(shear_left);
            break;
        case integrator_line:
            offset = getOffset(shear_right) + getCount(shear_right);
            break;
        default:
            offset = 0;
            break;

    }
    return offset;
}

int Object2D::getCount(drawing_instances instance){
    int vertexCount = 0;

    switch(instance){
        case outer_line:
            vertexCount = (int) outer_faces.size()*2;
            break;
        case outer_polygon:
            vertexCount = (int) outer_points.size();
            break;
        case inner_polygon:
            vertexCount = (int) inner_points.size();
            break;
        case outer_spring:
            vertexCount = (int) outer_springs.size() * 2;
            break;
        case inner_spring:
```

we added two parameters to the formal *Draw()* function of the object. One is the *shaderID* used for drawing and the other is *glm::mat4* type *WorldMatrix*. The shader id allows us to easily modify the color of the drawing instance, since we have programmed our shader to be able to accept a *glm::vec4* color input. The *worldMatrix* is used to define the translation, rotation, and scaling of the vertex. We will mention the role of *worldMatrix* again in the following chapters. In the *Draw()* function, we first call the *updateVertexArray()* function to update the vertex position and then draw different instances in the given order.

```cpp
//##ModelId=45F4D797017A
void Object2D::Draw(int shaderID, glm::mat4 worldMatrix)
{
    static bool findOnce = false;

    //  ===============  basic settings for ogl3 drawing
    updateVertexArray();
    glBindVertexArray(VAO);
//    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // set world matrix
    glUniformMatrix4fv(glGetUniformLocation(shaderID, "worldMatrix"), 1, GL_FALSE, &worldMatrix[0][0]);
    // get color location
    GLuint ColorLocation = glGetUniformLocation(shaderID, "colorChoice");
    glEnableClientState(GL_VERTEX_ARRAY);
    glPointSize(10);
    // ====================================================

    // drawing outer lines
    glUniform4f(ColorLocation, 0.5f, 0.2f, 0.5f, 0.7f);
    glDrawArrays(GL_LINES, getOffset(outer_line), getCount(outer_line));

    // drawing outer polygon
    glDrawArrays(GL_TRIANGLE_FAN, getOffset(outer_polygon), getCount(outer_polygon));

    // drawing inner polygon
    glUniform4f(ColorLocation, 0.0f, 1.0f, 0.0f, 0.5f);
    glDrawArrays(GL_TRIANGLE_FAN, getOffset(inner_polygon), getCount(inner_polygon));

    // drawing outer springs
    // (line)
    glUniform4f(ColorLocation, 0.2f, 0.2f, 1.0f, 1.0f);
    glDrawArrays(GL_LINES, getOffset(outer_spring), getCount(outer_spring));
    // (point) - need to be drawn using a loop to draw only one point, also the first point need to be red, c
    for(int i = 0; i < outer_springs.size(); ++i)
    {
        if(i == 0)
            glUniform4f(ColorLocation, 1.0f, 0.0f, 0.0f, 1.0f);
        else
            glUniform4f(ColorLocation, 1.0f, 1.0f, 0.0f, 1.0f);

        glDrawArrays(GL_POINTS, getOffset(outer_spring) + i*2, 1);
    }

    // drawing inner springs
    // (line)
    glUniform4f(ColorLocation, 0.2f, 0.2f, 1.0f, 1.0f);
    glDrawArrays(GL_LINES, getOffset(inner_spring), getCount(inner_spring));
    // (point)
    glUniform4f(ColorLocation, 1.0f, 0.0f, 0.0f, 1.0f);
    glDrawArrays(GL_POINTS, getOffset(inner_spring), getCount(inner_spring));

    // drawing radium springs
    glUniform4f(ColorLocation, 0.0f, 1.0f, 0.0f, 1.0f);
    glDrawArrays(GL_LINES, getOffset(radium_spring), getCount(radium_spring));
```

It is important to know that we draw the integrator line only when the drag force exists in the integrator, which is, generally speaking, when the user clicks the left mouse button. Thus the function of *glDrawArrays()* in the figure below needs to be placed inside this if statement.

```
    if(integrator != nullptr){
        if(integrator->dragExists)
        {
            if(findOnce == true)
            {
                FindClosestPoint();
                findOnce = false;
            }
            // draw integrator
            glUniform4f(ColorLocation, 1.0f, 1.0f, 0.0f, 1.0f);
            glDrawArrays(GL_LINES, getOffset(integrator_line), getCount(integrator_line));
        }
        else
        {
            FindClosestPoint();
            findOnce = true;
        }
    }
```

The content in this section takes object2D as example to introduce how to draw the vertices of the SOFTBODY dynamically. The next section will introduce our modernization of the main loop.

2.1.3 The Main Loop

The source code of main loop in SOFTBODY is based on the functionalities of GLUT, which the main feature is to register customize functions to some predefined call back functions in GLUT, then enter the main loop by using *glutMainLoop()* (as shown in the figure below).

```
int main(void)
{

    glutInitWindowPosition(200, 200);
    glutInitWindowSize( Width, Height);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);

//  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    main_window = glutCreateWindow("A Simulation Ball - Miao Song");

//    while(true){
    glutReshapeFunc(Reshape);

    glEnable(GL_BLEND);                                  // transparent
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);  // transparent
    glEnable(GL_DEPTH_TEST);

//  glutPassiveMotionFunc(motion);

    glutMouseFunc(Mouse);
//    Mouse();
    glutMotionFunc(Motion);
//    Motion();
    glutKeyboardFunc(Keyboard);
//    Keyboard();
    glutSpecialFunc(SpecialKeys);
//    SpecialKeys();

    glutDisplayFunc(Display);
//    Display();
    glutIdleFunc(Idle);
//    Idle();

    /*
    GLUI *glui = GLUI_Master.create_glui( "GLUI" );

    new GLUI_Checkbox( glui, "Wireframe", &wireframe );
    (new GLUI_Spinner( glui, "Segments:", &segments ))
        ->set_int_limits( 3, 60 );

    glui->set_main_gfx_window( main_window );
    GLUI_Master.set_glutIdleFunc( Idle );
    */
    glutMainLoop();
//    }
    return 0;
}
```

Since we do not have a profound knowledge on how the callback functions work, we decided to put the drawing, update, projection, binding and update of view matrix, and human interaction of the mouse into our main loop. The following figure shows the code of creating the shader, binding the initial projection, and the view matrix.

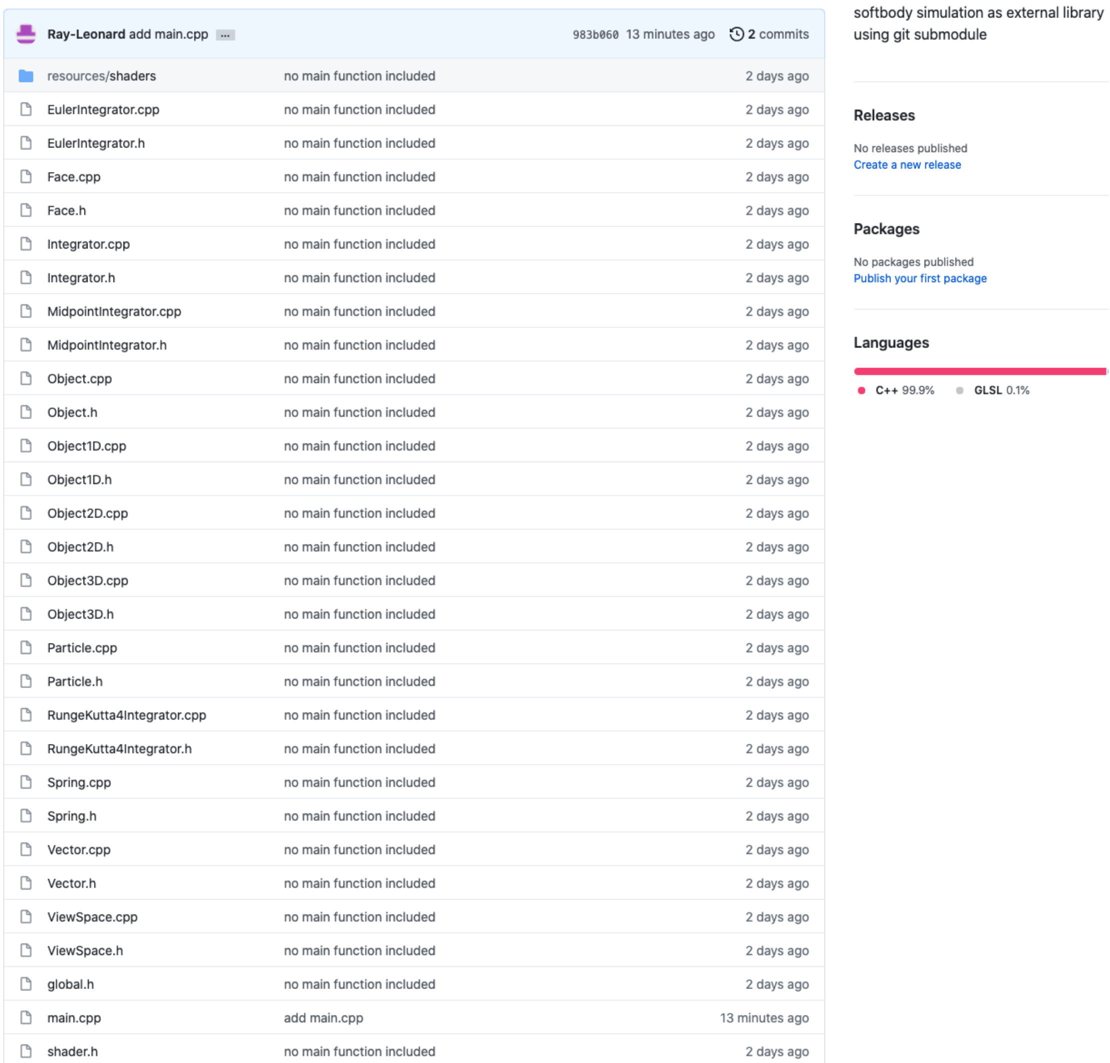
```cpp
// Compile and link shaders here ...
Shader shader("resources/shaders/core.vs", "resources/shaders/core.fs");
shader.use();
// ******************** end of system initializatoin ***************************************


// *****************Variables declarations*******************
float lastFrameTime = glfwGetTime();    // time is in seconds
float movementSpeed = 1.0f;
float initialEyeX = 0.0f;
float initialEyeY = 0.0f;
float initialEyeZ = 7.0f;
glm::vec3 initialEye(initialEyeX, initialEyeY, initialEyeZ);

int mousedown = 0;
float xMouse, yMouse;
// ******************* end of variable declaration***********


// ******************* set initial camera position *****************
glm::vec3 eye = initialEye;
glm::vec3 center(0.0f, 0.0f, 0.0f);
glm::vec3 cameraUp(0.0f, 1.0f, 0.0f);
glm::mat4 projectionMatrix = glm::perspective(glm::radians(70.0f), float(Width) / float(Height), 0.01f, 100.0f);
GLuint projectionMatrixLocation = glGetUniformLocation(shader.program, "projectionMatrix");
glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE, &projectionMatrix[0][0]);
glm::mat4 viewMatrix = glm::lookAt(eye,  // eye
    center,  // center
    cameraUp);
GLuint viewMatrixLocation = glGetUniformLocation(shader.program, "viewMatrix");
glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE, &viewMatrix[0][0]);

//**************** end of initial camera position *******************************************
```

The following figure is the code for creating, drawing, updating three SOFTBODY objects, and implementing integrator interaction.

```
    ViewSpace box;
    Object1D object1D;
    Object2D object2D;
    Object3D object3D;

    // set integrator type
    object1D.setIntegratorType(RK4);
    object2D.setIntegratorType(RK4);
    object3D.setIntegratorType(RK4);
    // main game loop
    while(!glfwWindowShouldClose(window))
    {
        // detect inputs from keyboard
        glfwPollEvents();

        // set background color (r, g, b, alpha)
        // can either do this inside the loop or outside the loop
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

        // Each frame, reset color of each pixel to glClearColor and clear depth buffer bit
        glClear(GL_COLOR_BUFFER_BIT);

        // compute the rotational angle every loop.
        float dt = glfwGetTime() - lastFrameTime;
        lastFrameTime += dt;



        //============== updating ===========
        // update mouse position
        double x = Width / 2.0, y = Height / 2.0;
        glfwGetCursorPos(window, &x, &y);
        xMouse = (4 * ((float)x / (float)Width)) - 2;
        yMouse = -((4 * ((float)y / (float)Height)) - 2);
        // update object
        object1D.Update(DT, mousedown != 0, xMouse, yMouse);
        object2D.Update(DT, mousedown != 0, xMouse, yMouse);
        object3D.Update(DT, mousedown != 0, xMouse, yMouse);

        // ============ drawing ======================
        box.Draw(shader.program);
        object1D.Draw(shader.program);
        object2D.Draw(shader.program);
        object3D.Draw(shader.program);

        // End of frame, used 2 buffers for displaying, this is swapping the buffer
        glfwSwapBuffers(window);
```

2.2 Forging SOFTBODY Simulation Code into a Library-Like Product

As mentioned in introduction, in order to allow other people who want to import softbody into their projects easily without copying all the files one by one to the source directory, we uploaded the code to a github repository and allowed others to use git submodule to create a pointer directly to the code in the cloud, while also modifying the softbody code locally to meet the needs of their projects.



Any user who needs to import SOFTBODY only needs to type *git submodule add* https://github.com/Ray-Leonard/softbody in the terminal for their git repository. The softbody code file can be imported locally, and then one only needs to set up the local IDE compilation environment to use it. Of course, one also need to make corresponding changes to your project's main function in order to successfully create and render a softbody object.

2.3 Brining SOFTBODY into PA2

To import SOFYTBODY to our PA2, after setting up the git submodule, we first need to hash include the corresponding header file and then compile and link SOFTBODY's shader. To locate the SOFTBODY objects on our stage, theoretically, we need to modify each vertex's position in the object manually. But in reality, thanks to the OpenGL 3 shader, we can directly modify the world matrix of the softbody vertex shader to make the softbody objects to render on top of the stage and rotate with the world following the previously defined arrow keys. This is why in Section 2.1.2, we mentioned that the *Draw()* function of the object needs to take in the *glm::mat4 worldMatrix*. Because of this, we only need to define a unique world matrix for SOFTBODY objects in the main function of our PA2, applying transformation and rotation to get the object in the right place without modifying the SOFTBODY source code. The figure below is the definition of the SOFTBODY matrix in our main functin.

```
// define a set of world matrix for softbody
glm::mat4 softbodyTranslateMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f,
    30*gridUnit, -35*gridUnit));
glm::mat4 softbodyMatrix = worldRotationMatrix * softbodyTranslateMatrix;
```

When we call the *Draw()* function, we only need to pass in the *softbodyMatrix* as a parameter.

```
object1D.Draw(sbShader.program, softbodyMatrix);
object2D.Draw(sbShader.program, softbodyMatrix);
object3D.Draw(sbShader.program, softbodyMatrix);
```

To make softbody objects bounce around naturally on our stage and in our scene, we have slightly modified the collision detection factors in both *global.h* and *Integrator.cpp* to make the stage the ground and constructed imaginary walls at the edges of the stage.

In fact, anyone who'd like to construct their own collision boundaries can easily modify these six variables in the following picture to make a collision box without changing anywhere else in the code.

```
// collision detection
struct Collision{
      float xl = -LIMIT-2.2;
      float xr = LIMIT+2.2;
      float yl = -LIMIT;
      float yr = LIMIT;
      float zl = -LIMIT+1;
      float zr = LIMIT-0.8;

};
```

Specifically, we made the collision factors to reside in a structure in *global.h* and made the *CollesionDetection()* function in *Integrator.cpp* to instantiate a collision structure, then proceed to use the factors in Collision structure to do boundaries checking.

```
//##ModelId=45F4D7980001
void Integrator::CollisionDetection(int i)
{
    bool changed = false;
    float decay = 0.5;
    Collision collision;

///////////////////check the outer layer particle if collide with the wall////////////////
        // Check #1 - X boundaries left
    if((object->outer_points[i]->r->x + object->outer_points[i]->dr->x) < collision.xl)
    {
        object->outer_points[i]->dr->x = collision.xl - object->outer_points[i]->r->x;
        changed = true;
    }    // Check #1 - X boundaries right
    else if((object->outer_points[i]->r->x + object->outer_points[i]->dr->x) > collision.xr)
    {
        object->outer_points[i]->dr->x = collision.xr - object->outer_points[i]->r->x;
        changed = true;
    }

    if(changed)
    {
        object->outer_points[i]->v->x = -decay * object->outer_points[i]->v->x;
        object->outer_points[i]->v->y =  0.5 * object->outer_points[i]->v->y;
        object->outer_points[i]->v->z =  0.5 * object->outer_points[i]->v->z;

    /*   if(object->dim == 1){
          if(i==1){
          object->outer_points[i]->r->x = 0;//object->outer_points[i]->r->x +
              object->outer_points[i]->dr->x;
          }

        }
        else{
        object->outer_points[i]->r->x = object->outer_points[i]->r->x + object->outer_points[i]->dr->x;
        }*/
        object->outer_points[i]->r->x = object->outer_points[i]->r->x + object->outer_points[i]->dr->x;
        changed = false;
    }
```

Eventually, since the left mouse button in PA2 controls the zoom-in and zoom-out of the camera, to allow the left mouse button to control the integrator, we added a mouse mode switch in the poll events area. After pressing R, we can use the mouse to control the camera, and pressing Y allows user to control the integrator (code shown in the figure below).

```
// *************** handle inputs *************************
if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
{
    glfwSetWindowShouldClose(window, true);
}

// *************** for changing mouse mode ***********
if(glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS)
{
    mouseMode = 0;  // to control camera
}

if(glfwGetKey(window, GLFW_KEY_Y) == GLFW_PRESS)
{
    mouseMode = 1;  // to control softbody
}

// ************ integrator drag forces *************
if(glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS && mouseMode == 1)
{
    mousedown = 1;
    double x , y;
    glfwGetCursorPos(window, &x, &y);
    xMouse = (4 * ((float)x / (float)Width)) - 2;
    yMouse = -((4 * ((float)y / (float)Height)) - 2);
}
else{
    mousedown = 0;
}
//Right click ( use mouse movement in x direction to pan )
if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_RIGHT) == GLFW_PRESS && mouseMode == 0) {

        glfwGetCursorPos(window, &mousePosX, &unusedMouse);
        dx = mousePosX - lastMousePosX;
        center -= glm::vec3( dx * 0.05f,0.0f,0.0f);
        lastMousePosX = mousePosX;
        lastMousePosY = unusedMouse;
}
else {
        dx = 0.0f;
}
```

In this way, we can interact with imported SOFTBODY objects in the PA2 scene (the picture below is a screenshot of the project running).


COMP371 PA2
LE48

3. PA3 Function Key Documentation

This section will document the function keys of our PA3, serving as a readme file. Most function keys are inherited from PA2, a few are added for PA3.

Camera movements

- Press “R” to use the following camera movements
- Right click and drag - move in x direction to pan by moving mouse
- Middle click and drag - move in y direction to tilt by moving mouse
- Left click and drag - move into / out of scene by moving mouse

Dragging Soft Bodies Around

- Press “Y” to switch to soft body controller
- Left click and drag - apply integrator forces to soft body objects

World Rotation

- up - counter-clockwise rotation of world around y axis
- down - clockwise rotation of world around y axis
- right - clockwise world rotation around z axis
- left - anti-clockwise world rotation around z axis

Model selection and Model manipulations

- 1 - select 1st set of model "JI40" to manipulate
- 2 - select 2nd set of model "LE48" to manipulate
- 3 - select 3rd set of model "ZN46" to manipulate
- 4 - select 4th set of model "YN40" to manipulate
- 5 - select 5th set of model "FG47" to manipulate
- 0 - reset camera/models/world rotations back to initial position

- (once a model is selected, the following manipulation keys will only affect the selected model)
- W (shift + w) - move model forward
- S (shift + s) - move model backward
- A (shift + a) - move model left
- D (shift + d) - move model right
- U (shift + u) - scale model up
- J (shift + j) - scale model down
- u - scale model up by scaleDelta each press (defined in program)
- j - scale model down by scaleDelta each press (defined in program)
- a - rotate model counter-clockwise around its y axis
- d - rotate model clockwise around its y axis
- q - rorate model counter-clockwise by rotateDelta (defined in program) each press
- e - rotate model clockwise by rotateDelta (defined in program) each press
- z - continuous forward movement around vertical shear transformation
- Z (shift + z) - continuous forward movement around lateral shear transformation
- m - continuous forward movement around vertical shear transformation
- M (shift + m) - continuous forward movement around lateral shear transformation
- v - small movement(cos(10)) around vertical shear transformation
- n - small movement(cos(10)) around vertical shear transformation
- c - small movement(cos(10)) around lateral shear transformation
- b - small movement(cos(10)) around lateral shear transformation

Other Switches

- x - toggling texture on/off
- o - toggling shadow on/off
- t - render triangles mode
- l - render lines mode
- p - render points mode
- Space - place all models randomly on the grid

4. CONCLUSION

In this chapter, we will briefly summarize the knowledge we have learned from PA3. Through modernizing the source code of SOFTBODY, we first learned the difference between OpenGL 2 and OpenGL 3 when rendering graphics. Although the rendering code of OGL 2 is simpler than OGL 3, it is not as powerful as OGL 3 and does not have the same advanced degrees of freedom as OGL 3. This is because of OGL 3's programmable shader. Moreover, we have learned how to render real-time changing graphics by using dynamic buffers. It deepened our understanding of the OGL 3 rendering pipeline. This is also one of the most significant gains of the project. In completing the modernization, we also learned about the composition of the SOFTBODY, which itself is composed of various springs. Each spring has its own force. A SOFTBODY is an entire structure supported by networks of springs. SOFTBODY also reacts to external forces, such as the pressure of colliding with the wall and the pulling force generated by the integrator, resulting in deformation and displacement. Adjusting the SOFTBODY parameters can change the sensitivity to external forces and stability. When importing SOFTBODY into our PA2, we learned to use git submodule to avoid unnecessary copying of files but can directly access the source code in the repository. At the same time, we also learned how to minimally modify the source code of SOFTBODY to run with our PA2.

Thanks for providing us with a chance to work on such a challenging project. We have learned a lot.

5. REFERENCE

[1] Miao Song, Serguei A. Mokhov, and Peter Grogono. 2009. Designing an interactive OpenGL slide-based presentation of the softbody simulation system for teaching and learning of computer graphics techniques. In Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering (C3S2E '09), Bipin C. Desai and Carson K. Leung (Eds.). ACM, New York, NY, USA, 131-136. DOI=10.1145/1557626.1557647

[2] Miao Song and Peter Grogono. 2009. Application of advanced rendering and animation techniques for 3D games to softbody modeling and animation. In Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering (C3S2E '09), Bipin C. Desai and Carson K. Leung (Eds.). ACM, New York, NY, USA, 89-100. DOI=10.1145/1557626.1557640

[3] Song, Miao. "Dynamic deformation of uniform elastic two-layer objects.", Master's thesis, arXiv preprint arXiv:0907.4364 (2009).

[4] Serguei A. Mokhov et al. OpenISS – SoftbodySimulationSystem. [online],[accessed 6-May-2021],2016-2020.https://github.com/OpenISS/SoftbodySimulationSystem