CS475 COMPUTER GRAPHICS

# Course Project

Parsana Vivek Navinbhai (210010047)

## Introduction

In this project, we explore the fascinating world of particle systems, which are essential in simulating various natural and physical phenomena. Particle systems consist of a large collection of particles that move according to predefined physical laws and respond to external forces. These systems are widely used in computer graphics to create realistic simulations of effects like fire, explosions, smoke, and more.

The second part of the project transitions to hierarchical modeling, a technique used to animate complex structures with interconnected parts. This simulation features a transformer model, where each segment (like arms or legs) can be individually controlled and animated. The hierarchical structure allows for intricate, lifelike animations, where movements in one part of the model influence all connected parts. The project leverages OpenGL for real-time rendering and user interaction, bringing the animated transformer to life.

## Explanation of First Part of Code

### a1sim.cpp

We put whole the code in src folder where all supported cpp and hpp file present. The outside src folder there are MakeFile which use to compile and make a executable bin file from all the files in src folder. I write detail instruction to how to run code in your compiler in the Readme.

Now in src folder there are a a1sim.cpp which is main file and convert to executable .exe file on running the code in terminal. Now what is inside the a1sim.cpp?

```
int getT(){
  time_t tim = time(0);
  tm *gottime = localtime(&tim);
  int sec = gottime->tm_sec;
  int min = gottime->tm_min;
  int hr = gottime->tm_hour;
```

This is code to get current time of running system. This capture the local time of computer. Then I convert the whole time in second and return it. So basically getT() function return the current time in seconds.

Now we move toward the main function of file.

```
GLFWwindow* window;

glfwSetErrorCallback(csX75::error_callback);

if (!glfwInit())
  return -1;

glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

window = glfwCreateWindow(1000, 1000, "CS475 Assignment", NULL, NULL);
if (!window)
  {
    glfwTerminate();
    return -1;
  }

glfwMakeContextCurrent(window);

glewExperimental = GL_TRUE;
GLenum err = glewInit();
if (GLEW_OK != err)
  {
    std::cerr<<"GLEW Init Failed : %s"<<std::endl;
  }

glfwSetKeyCallback(window, csX75::key_callback);
glfwSetFramebufferSizeCallback(window, csX75::framebuffer_size_callback);
```

This code is basically create a window to visualise a animation in window.

Now if window is not created or there are any kind of specific error then this error callback a below function.

```
void error_callback(int error, const char* description)
{
  std::cerr<<description<<std::endl;
}
```

If not error then code continue and check the glfw intialise or not. If not intialise then return -1.

Then glfw check for opengl and anyother supported version.

After that glfw create a window of size 1000*1000 with name of CS475 Assignment.

Now again it check for window is created or not, if not then return -1.

After that it check for glew is OK or not.

After all above procedure done then we setback some basic keys like on press ESC terminate keys. To implement this the code call the setkeycallback() function, which is present in gl_frameword.cpp . and I also attest below.

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
  //!Close the window if the ESC key was pressed
  if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
    glfwSetWindowShouldClose(window, GL_TRUE);
}
```

After that it callback resize window for glfw. Which resize the whole glfw window to 1000*1000.

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
  //!Resize the viewport to fit the window size - draw to entire window
  glViewport(0, 0, width, height);
}
```

Now we are done with all intialisation procedure. So we are starting with the main animation.

```
csX75::initGL();
ParSys* newSys = new ParSys();
ParSys2* newSys2 = new ParSys2();
newSys->initShaderGL();
newSys->initVBSHGL();
bool flag = 1;
bool play = 1;
```

So first we intialise the GL function which was present in framework_gl. I attest a code below for same.

```
void initGL(void)
{
  //Set framebuffer clear color
  glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
  //Set depth buffer furthest depth
  glClearDepth(1.0);
  //Set depth test to less-than
  glDepthFunc(GL_LESS);
  //Enable depth testing
  glEnable(GL_DEPTH_TEST);
  //Enable Gourard shading
  glShadeModel(GL_SMOOTH);
}
```

Which was simply initialise the basic functionality of GL.

Then we create a two object name newSys and newSys2 which are the particle system 1 and particle system 2 respectively. We look in detail about particle systems later. Then we intialise the shader and VBSHGL for first system because we want then when we execute the code the first system must run. So we initialise the first system. We see the code for

both later, here just understand that newSys are initialised. Then we create two bool variable flag and play. Flag is 1 if first system is in running condition and 0 if second animation is in running condition. Play is check for both system is in play or pause conditions.

Then we have while loop which runs till we not close the window. To close the window we have two option, first one is to press esc and another one is to clode the window.

Now in for loop there are whole code to rander the both animation.

So here is explanation to rander things.

```
if(glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS) {
  newSys->initShaderGL();
  newSys->initVBSHGL();
  flag = 1;
}
else if(glfwGetKey(window, GLFW_KEY_2) == GLFW_PRESS) {
  newSys2->initShaderGL2();
  newSys2->initVBSHGL2();
  flag = 0;
}
```

This code simply shift between both animation by pressing 1 and 2. If you click 1 then the 1st particle system is intialise and mark flag=1. If you press 2 then 2nd particle system initiaise and mark flag=0 as discussed earlier. Now particle system 2 intishader2() and initVBSHGL2() is nearly same as particle system1, but there are slight difference which we discussed later.

```
else if(glfwGetKey(window, GLFW_KEY_P) == GLFW_PRESS){
    play = 0;
    int curT = getT();
    if(flag){
      newSys->pause(curT);
    }
    else{
      newSys2->pause2(curT);
    }
}
else if(glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS){
    play = 1;
    if(flag){
      newSys->pau = 0;
    }
    else{
      newSys2->pau2 = 0;
    }
}
```

This is code for play and pause. If key P is pressed then the system need to pause. Now there are one question which simulation need to pause? If flag=1 means system 1 is running, So we pause system 1. Else pause system 2. But how? We have pause function in out system class and we pass time as argument and we see details of pause function later.

Same for resume, but here we don't have function, we simply have one variable in system of particle which check the system is paused or not. We see uses of this variable later on.

Now we are at position to rander the both system.

Simple if flag=1 then we have to rander system 1 else we have to rander system2. Code for system 1 is below.

```
if(flag){
  newSys->renderGL();

  glfwSwapBuffers(window);

  glfwPollEvents();

  if(!newSys->pau){
    for(int i=0;i<newSys->storage.size();i++){
      if(getT() - newSys->storage[i]->timeOrigin >= 3.2){
        newSys->storage.erase(newSys->storage.begin()+i);
        i--;
      }
    }
  }
}
```

Here if flag = true then we simply rander the newSys (We see detail explanation of this function later on). Now to make simulation faster there are two window at time in glfw. When top window is running, at same time below window is starting rendering. So two window is useful. So with help of swapBuffer we swap both window. After that we use the variable which was change at time of play, a pause variable in particle system. If system is not paused then we check that any particle currently present on screen is out of time. Means any particle have life time greater than 3.2 for 1st system then we simply delete this particle. Here storage is vector<int> type which simply store all present particle on screen. We discuss about this later on. This for loop simply checks for all particle.

Now after that we have else condition means if system 2 is running. Code for same is below. Code is same for both system. But system 2 calls in there respective class functions rather than system 1 class function. And also check if particle is outof time and not paused.

So code for same attest below,

So this is how both system works.

```
else{
  newSys2->renderGL2();

  glfwSwapBuffers(window);

  glfwPollEvents();

  if(!newSys2->pau2){
    for(int i=0;i<newSys2->storage2.size();i++){
      if(getT() - newSys2->storage2[i]->timeOrigin2 >= 3.9){
        newSys2->storage2.erase(newSys2->storage2.begin()+i);
        i--;
      }
    }
  }
}
```

This is end of a1sim.cpp.

## Particle.h / particle.cpp

This is simulation code for system 1.

```
class Particle{
    public:
    glm::vec2 position,velocity;
    int timeOrigin;
    int timeTotal = 3;
    int n = 200;
    Particle(float angle,int t);
};
```

First we include some basic libraries which useful in execution. Then we create a class name Particle which has argument position,velocity which are vector of 2 means have x,y.

Now then we have timeOrigin which stores creation time of particle which useful at time of destroy. Then we have timeTotal which stores total leavable time of particle in system. Here it is 3 second. Then we have n which are number of steps which useful at time of integrate system. Then we have initialization of particle with angle and t. We see detail about both below.

```
#include"particle.h"

Particle::Particle(float angle,int t){
    position = {0.0f,-2.0f};
    velocity = {(float)2.5*cos(angle),(float)2.5*sin(angle)};
    timeOrigin = t;
}
```

The initialization of particle class. Have angle and t as input. Then we intialise particle at position at 0,-2 meas at middle bottom of screen. Then We intialise velocity of particle as 2.5cos(angle), 2.5sin(angle). We decide 2,5 because by setting this figure particle not leaving the screen and simulation looks cool!

## particleSystem.h / particleSystem.cpp

This is code for particle system of simulation 1.

There are some variable and functions in .h . first is vector of storage which stores all the particle present in class at a current time. There are some reasons to choose a vector because any particle is accessible in O(1). We can simply erase any particle in O(n). to insert particle we need only O(1). if we use priority queue sort on basis of time then we have to update a priority queue at every point, which takes O(nlogn) which was so time consuming. So vector is better choice. Then we declare some vertex buffer and verted array object.

We also create a variable for shaderprogram and also to store the matrix which useful in transform coordinate system. And then we have some basic functions which we discussed just after that.

Below is code for above explanation.

```
#ifndef _PARTICLESYSTEM_HPP_
#define _PARTICLESYSTEM_HPP_

#include"particle.h"
#include"vector"
using namespace std;
class ParSys{
    public:
    vector<Particle*> storage;
    GLuint shaderProgram;
    GLuint vbo, vao;
    glm::mat4 view_matrix;
    glm::mat4 ortho_matrix;
    glm::mat4 modelviewproject_matrix;
    GLuint uModelViewProjectMatrix;
    bool pau;
    void initShaderGL();
    void move();
    void initVBSHGL();
    void renderGL();
    void pause(int t);
};
#endif
```

### initShaderGL()

Below is code for initShaderGL(). First we clear all storage array which useful when we shift between animation. Then we load shader.glsl and vertex.glsl and we discuss about both file at end of report. Then we create a shaderlist and push back both shaderfile in shaderlist. Then we create a shaderProgram fromshader pogram of namespace use in csX75 of shader_util.cpp. Then we ask GL for vbo and then set it as current buffer. Then follow same for vao. Then we enable the vertex attribute object. And then we define layout for rander. In glVertexAttribPointer the first element is of layout=0, second is of how many element in

coordinate of current particle means we have x,y,z, So here it is 3. It is float so we write GL_FLOAT as 3rd argument. And last two are about is there any buffer or not. And done.

```
void ParSys::initShaderGL(){
  storage.clear();
  std::string vertex_shader_file("src/vshader.glsl");
  std::string fragment_shader_file("src/fshader.glsl");
  std::vector<GLuint> shaderList;
  shaderList.push_back(csX75::LoadShaderGL(GL_VERTEX_SHADER, vertex_shader_file));
  shaderList.push_back(csX75::LoadShaderGL(GL_FRAGMENT_SHADER, fragment_shader_file));
  shaderProgram = csX75::CreateProgramGL(shaderList);
  //Ask GL for a Vertex Buffer Object (vbo)
  glGenBuffers (1, &vbo);
  //Set it as the current buffer to be used by binding it
  glBindBuffer (GL_ARRAY_BUFFER, vbo);
  glGenVertexArrays (1, &vao);
  //Set it as the current array to be used by binding it
  glBindVertexArray (vao);
  //Enable the vertex attribute
  glEnableVertexAttribArray (0);
  //This the layout of our first vertex buffer
  //"0" means define the layout for attribute number 0. "3" means that the variables are vec3 made from every 3 floats
  glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
}
```

## initVBSHGL()

In the starting phase of this function, we create a 13 particle and releases in different direction like at 80°, 81.25°, 82.5°, 83.75°, 85°, 87.5°, 90°, 92.5°, 95°, 96.25°, 97.5°, 98.75°, and 100° with positive x axis. We also push this particle in storage because it is present in current particle system.

Now we have to rander this point, So second image is about how to rander point. Now we consider point as triangle. We discuss after some time why we choose triangle. Now all particle present in storage has some x,y coordinate, we have to consider this point as centroid of triangle and make all coordinate of equilateral triangle from this x,y coordinate. To derive all three coordinate from x,y is simple high school math. Here we consider a length of each side of triangle as 0.008 because by considering this length the triangle looks like particle. So with for loop of storage.size() iteration we convert all particle to triangle and store into one another array of name point to rander it. Now after that point array is of size storage.size()*9, so we create a buffer of that size. Now last element of glBufferdata is GL_DYNAMIC_DRAW indicates we have some kind of dynamic simulation rather than static.

```
void ParSys::initVBSHGL()
{
  Particle* temp1 = new Particle(1.39626,getTime());
  storage.push_back(temp1);
  Particle* temp2 = new Particle(1.48353,getTime());
  storage.push_back(temp2);
  Particle* temp3 = new Particle(1.65806,getTime());
  storage.push_back(temp3);
  Particle* temp4 = new Particle(1.74533,getTime());
  storage.push_back(temp4);
  Particle* temp5 = new Particle(1.439897,getTime());
  storage.push_back(temp5);
  Particle* temp6 = new Particle(1.701696,getTime());
  storage.push_back(temp6);
  Particle* temp7 = new Particle(1.57,getTime());
  storage.push_back(temp7);
  Particle* temp8 = new Particle(1.422443,getTime());
  storage.push_back(temp8);
  Particle* temp9 = new Particle(1.719149,getTime());
  storage.push_back(temp9);
  Particle* temp10 = new Particle(1.684243,getTime());
  storage.push_back(temp10);
  Particle* temp11 = new Particle(1.45735,getTime());
  storage.push_back(temp11);
  Particle* temp12 = new Particle(1.527163,getTime());
  storage.push_back(temp12);
  Particle* temp13 = new Particle(1.61443,getTime());
  storage.push_back(temp13);
```

Now why we take particle as triangle?

If we take a circular object then to rander a circle we need a around 50-100 triangle combination which was a quite complex. So for every particle we need this amount of triangle. So better is to rander just 1 triangle for a one particle.

```
  float point[storage.size()*9];
  int m = 0;
  for(int i=0;i<storage.size();i++){
    float x = storage[i]->position[0];
    float y = storage[i]->position[1];
    point[m++] = (float)(x);
    point[m++] = (float)(y + (0.008/sqrt(3)));
    point[m++] = 0.0f;
    point[m++] = (float)(x+(float)(0.008/2));
    point[m++] = (float)(y-(float)(0.008/(2*sqrt(3))));
    point[m++] = 0.0f;
    point[m++] = (float)(x-(float)(0.008/2));
    point[m++] = (float)(y-(float)(0.008/(2*sqrt(3))));
    point[m++] = 0.0f;
  }
  glBufferData (GL_ARRAY_BUFFER, storage.size()*9*sizeof (float), point, GL_DYNAMIC_DRAW);
}
```

## randerGL()

```
void ParSys::renderGL(void)
{
  if(!pau){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    move();
    initVBSHGL();
    glUseProgram(shaderProgram);

    glBindVertexArray (vao);

    view_matrix = glm::lookAt(glm::vec3(0.0,0.0,-2.0),glm::vec3(0.0,0.0,0.0),glm::vec3(0.0,1.0,0.0));

    ortho_matrix = glm::ortho(-2.0f, 2.0f, -2.0f, 2.0f, -20.0f, 20.0f);
    modelviewproject_matrix = ortho_matrix * view_matrix;

    glUniformMatrix4fv(uModelViewProjectMatrix, 1, GL_FALSE, glm::value_ptr(modelviewproject_matrix));

    glDrawArrays(GL_TRIANGLES, 0, storage.size()*3);
  }
```

```
else{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glUseProgram(shaderProgram);

  glBindVertexArray (vao);

  view_matrix = glm::lookAt(glm::vec3(0.0,0.0,-2.0),glm::vec3(0.0,0.0,0.0),glm::vec3(0.0,1.0,0.0));

  ortho_matrix = glm::ortho(-2.0f, 2.0f, -2.0f, 2.0f, -20.0f, 20.0f);
  modelviewproject_matrix = ortho_matrix * view_matrix;

  glUniformMatrix4fv(uModelViewProjectMatrix, 1, GL_FALSE, glm::value_ptr(modelviewproject_matrix));

  glDrawArrays(GL_TRIANGLES, 0, storage.size()*3);
}
```

Now this is rander function. So we have two condition like system is paused or not. So different between both is only that if system is paused we don't move a system and not create a new particle. All other thing for both is same. We look upon this. So with glClear we clear the present screen. Then we use shaderProgram which initialise earlier. Then we bind vertex array vao. Then we create a view matrix which means at which position our camera (eye), center and up vector located. So we define like eye at 0,0,-2, the centre at 0,0,0 and up vector is 0,1,0. Then we create a orthogonal matrix means dimensions or range of system. The argument of ortho is left,right,top,bottom,up,down. So we define system from -2 to 2 in x and y direction. And define -20 to 20 in z direction. Then we multiply both matrix to make view projection. And then initialises it. Now after all above initialization process we simply draw all triangle. Number of triangle in system is number of particle is equal to storage.size(). So total points is storage.size()*3 and finally we draw all triangle.

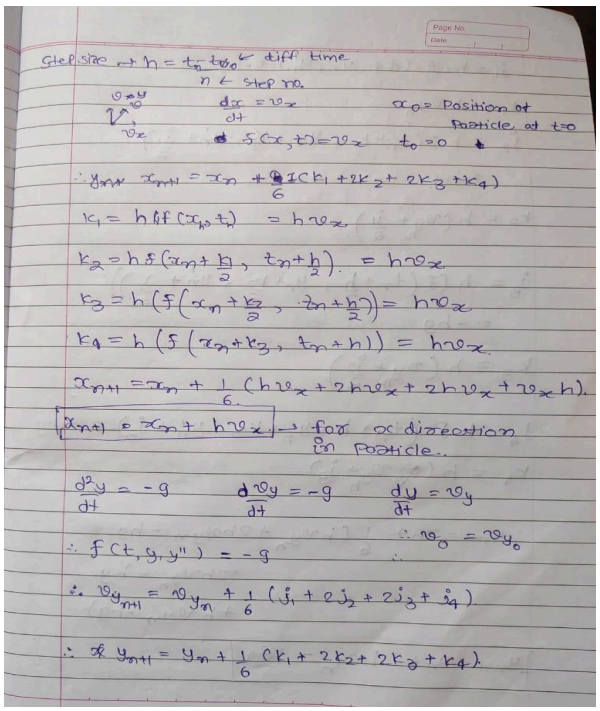
## pause()

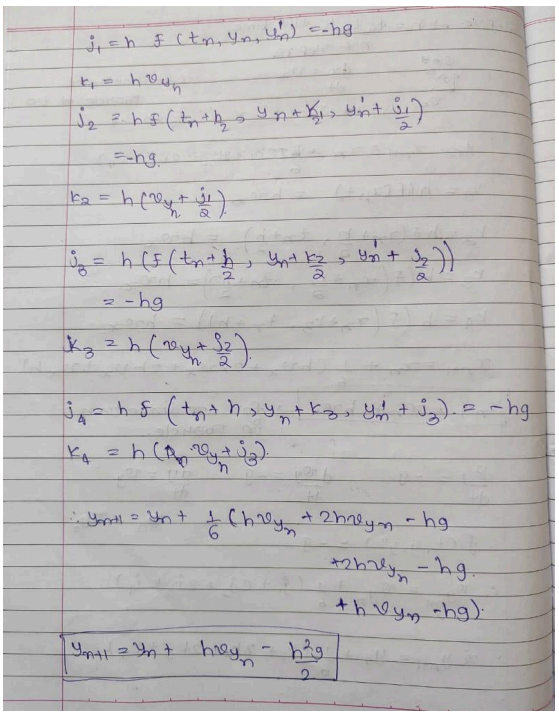
This is code for pause system when pause is call on a1sim.cpp. You can see reference of pause call on a1sim.cpp text. So first we make pause bool equal to 1. The argument of function is current time of system. Then we change time of origin of all system particle to current time t because it already simulated time t before pause function call. So we change it and done about pause. All other thing of pause is done in a1sim.cpp.

```
void ParSys::pause(int t){
  pau = 1;
  for(int i=0;i<storage.size();i++){
    storage[i]->timeOrigin = t;
  }
}
```

## move()

```
void ParSys::move(){
  for(int i=0;i<storage.size();i++){
    Particle* p = storage[i];
    float h= (float)p->timeTotal/(float)p->n;
    p->position[0]= p->position[0]+(p->velocity[0]*h);
    p->position[1]= p->position[1]+(h*p->velocity[1])-(h*h*0.49);
    p->velocity[1]= p->velocity[1]-(h*0.98);
  }
}
```





## Particle2.h / particle2.cpp

Particle2.h is same as particle.h, all code and explanation is same.

But in particle2.cpp we have some extra code. In particle.cpp we intialise all particle from one point 0,-2, but here we rander at from different position which ranges from -2,-1.9 to -2,1.9 with regular interval of 0.2. So we have switch condition for all cases. In particle.cpp we have different velocity for all particle depend on angle but here velocity is same 0,0 for all particle, means we don't need to initialise velocity.

## particleSystem2.h / particleSystem2.cpp

Code for particle system 2. Where all argument are same but there are additional 2 at end of all name to avoid duplications. So below is functions present in .h .

### initShaderGL2()

Code is same as initShaderGL().

### initVBSHGL2()

```
for(int i=1;i<=21;i++){
  Particle2* temp = new Particle2(getTime2(),i);
  storage2.push_back(temp);
}
```

Here first we initialise all 21 particle and push back in current storage.

Then according to second image we create a point vector which stores all coordinates of triangle. Now how many triangles are there? So for storage.size() particle there are storage.size() number of triangle. What is this extraa 100? Now this simulation is about the fluid flow around cylinder. So we have to also draw a top view of cylinder which is circular. So to rander circle we take 100 triangle. Now after that first for loop of storage.size() iteration is to create triangle from particle as discussed in system 1. Now another for loop of size steps = 100 is two create a circle. Here to find all coordinate of circle with given angle is also high school maths. And after that all coordinates of triangle stores in point array. So now we store all points in current buffer. And done.

```
float point[(storage2.size()+100)*9];
int m = 0;
for(int i=0;i<storage2.size();i++){
  float x = storage2[i]->position2[0];
  float y = storage2[i]->position2[1];
  point[m++] = (float)(x);
  point[m++] = (float)(y + (0.008/sqrt(3)));
  point[m++] = 0.0f;
  point[m++] = (float)(x+(float)(0.008/2));
  point[m++] = (float)(y-(float)(0.008/(2*sqrt(3))));
  point[m++] = 0.0f;
  point[m++] = (float)(x-(float)(0.008/2));
  point[m++] = (float)(y-(float)(0.008/(2*sqrt(3))));
  point[m++] = 0.0f;
}
for(int i=0;i<steps;i++){
  float newX = radius*sin(angle*i);
  float newY = radius*cos(angle*i);
  point[m++] = 0.0f;
  point[m++] = 0.0f;
  point[m++] = 0.0f;
  point[m++] = prevX;
  point[m++] = prevY;
  point[m++] = 0.0f;
  point[m++] = newX;
  point[m++] = newY;
  point[m++] = 0.0f;
  prevX = newX;
  prevY = newY;
}
glBufferData (GL_ARRAY_BUFFER, m*sizeof (float), point, GL_DYNAMIC_DRAW);
```

### randerGL2()

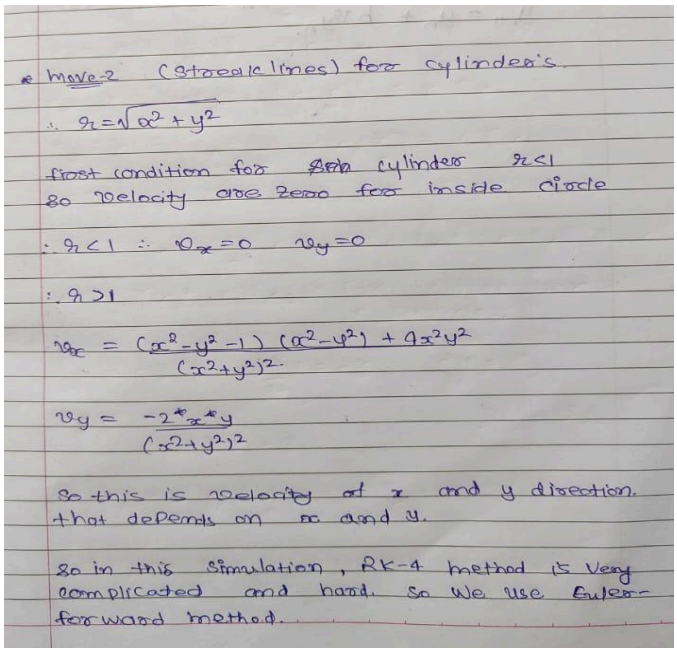
Code is almost same a randerGL(). We have extra triangle of cylinder at center of screen. So I added some extraa coordinated in glDrawArray().

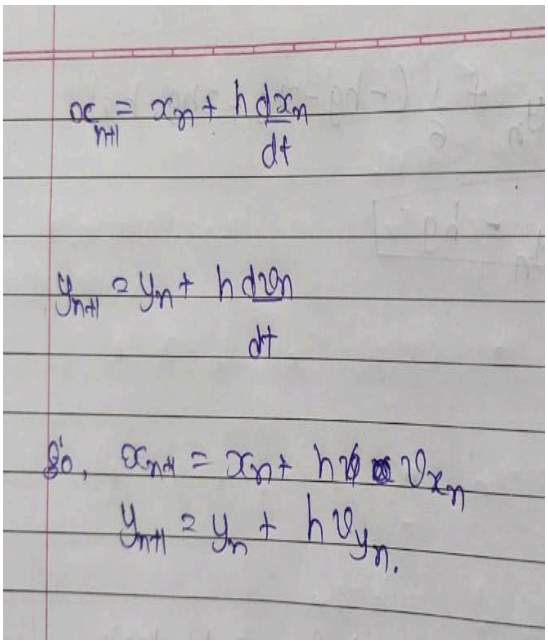
### pause2()

Same as pause().

### move2()

```
void ParSys2::move2(){
  for(int i=0;i<storage2.size();i++){
    Particle2* p = storage2[i];
    float h = (float)((float)p->timeTotal2/(float)p->n2);
    float x=(float)p->position2[0];
    float y=(float)p->position2[1];
    float r= pow(pow(x,2)+pow(y,2),0.5);
    if(r<1){
      p->velocity2[0]=0;
      p->velocity2[1]=0;
    }
    else{
      p->velocity2[0]=((pow(x,2)-pow(y,2)-1)*(pow(x,2)-pow(y,2)) + (4*x*x*y*y))/pow(r,4);
      p->velocity2[1]=-2*x*y/pow(r,4);
    }
    p->position2[0] += (p->velocity2[0]*h);
    p->position2[1] += (p->velocity2[1]*h);

  }
}
```




So now we are done with both system code.

Now we see the code of shader files.

## Vshader.glsl

```
#version 330

in vec3 vp;
uniform mat4 ModelViewProjectMatrix;

void main ()
{
  gl_Position = ModelViewProjectMatrix*vec4(vp,1.0);
}
```

This is simple. We have input argument is 3d position of triangle. We initialise a mat4 and then convert the 3d triangle coordinate to homogenous coordinated by adding 1 and then multiply it with mat4 ModelViewProjectionMatrix to project the screen on required direction. And return the resultant coordinate as gl_position. So this is basic functionality of vertex shader.

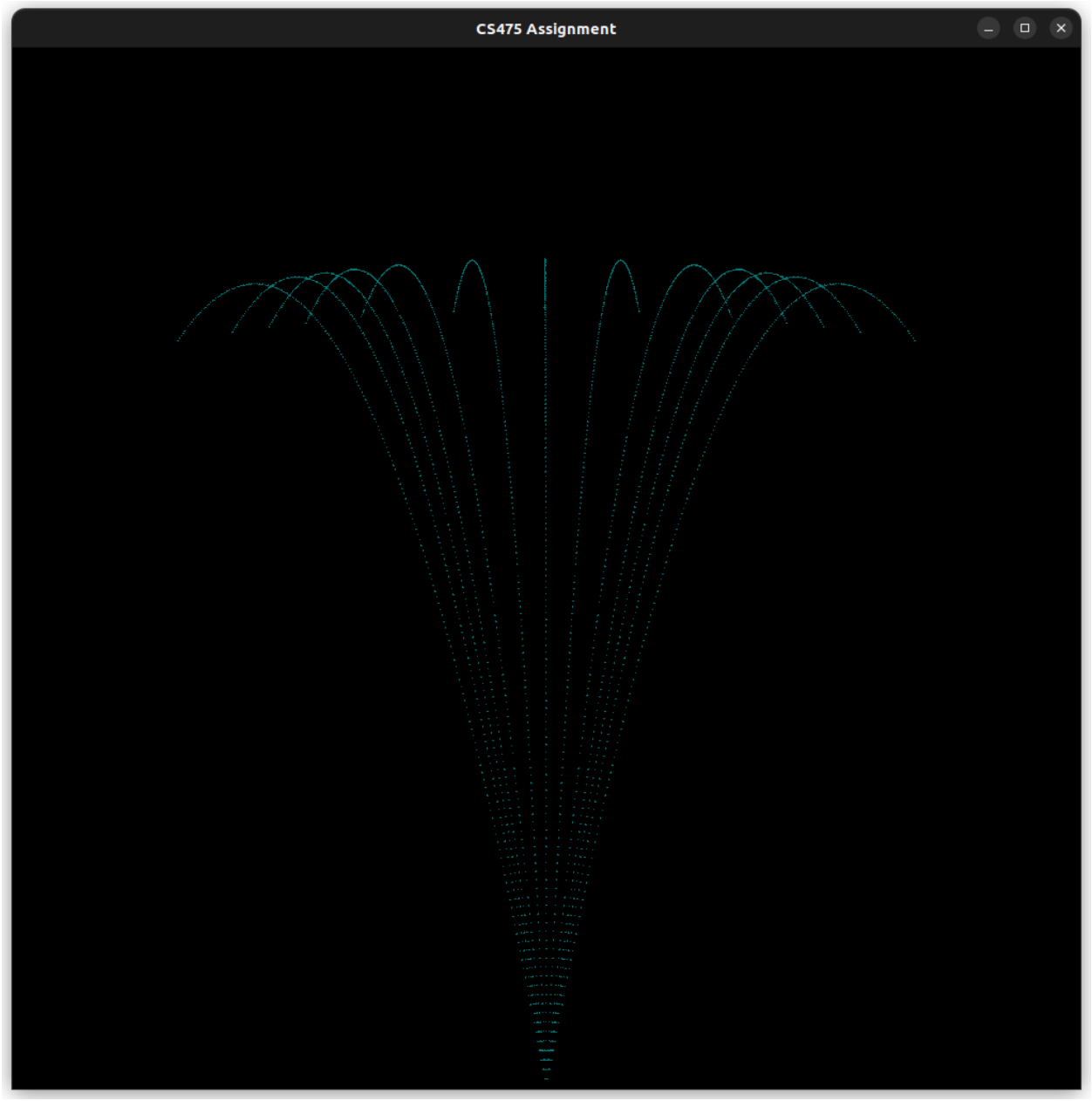
## Fshader.glsl

```
#version 330

out vec4 frag_colour;

void main ()
{
  frag_colour = vec4 (0.0, 0.458, 0.466, 1.0);
}
```

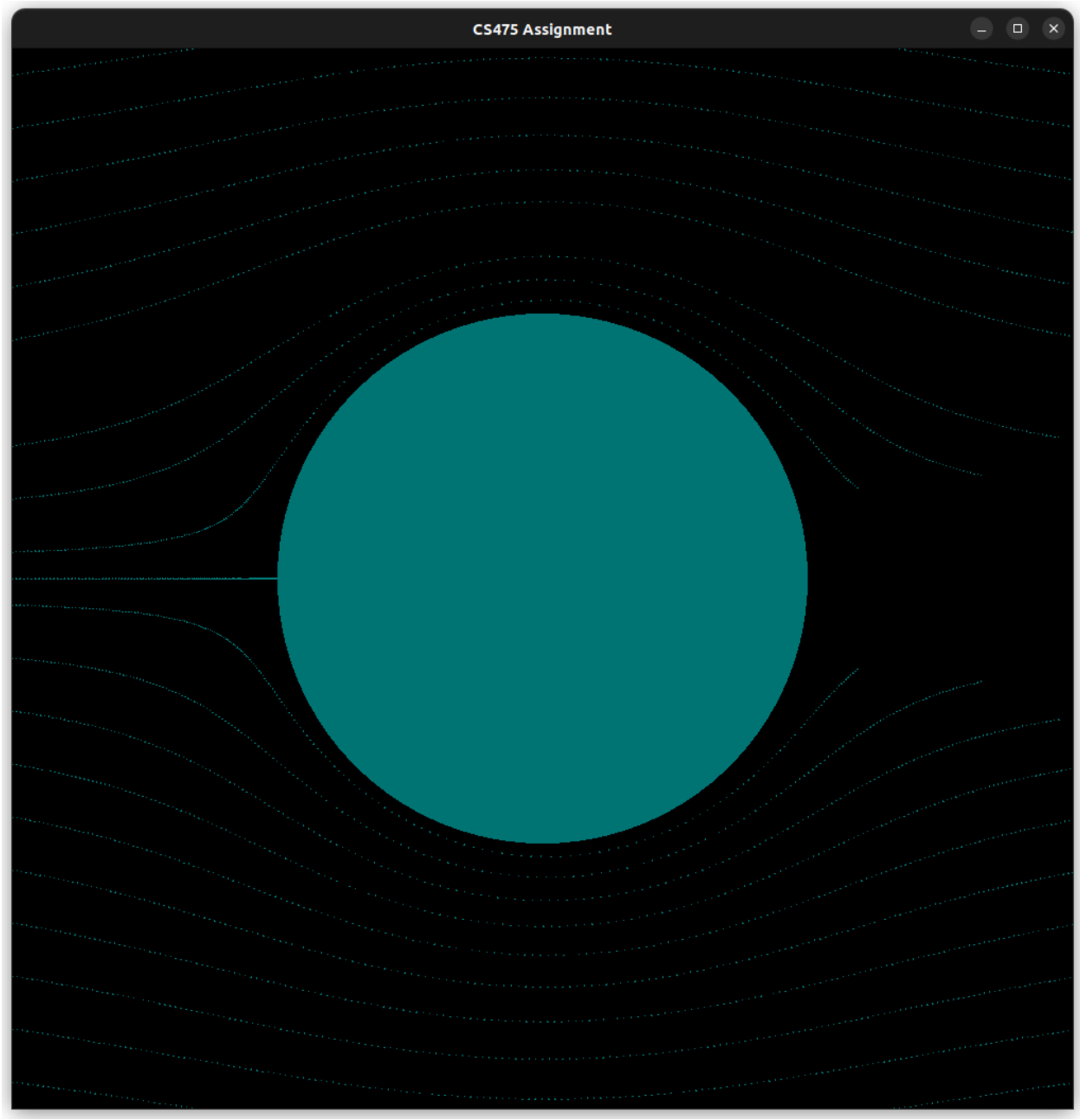
Here we simply intialise colors of all particle. The given is of color water particle in RGB system. And done we return it.

So now we done with all code. So it is time to run code and see what happens!!

## Final output




System 1 - sprinklr system




System 2 - fluid flow around non viscid, non lifting cylinder.

# Explanation of Second Part of Code

## Running the Code

- The project requires an OpenGL driver supporting version 4.1 or higher. For systems supporting OpenGL 3.2+, minor code modifications are necessary.
- Instructions for modifying the OpenGL version in the code are provided.

## Key Functions:

1. **Hierarchical Modeling (`07_hierarchical_modelling.cpp` & `07_hierarchical_modelling.hpp`):**
    - `setPositions(double xmin, double xmax, double ymin, double ymax)`: Sets the positions for the vertices.
    - `void quad(int a, int b, int c, int d, glm::vec4 v_positions[], glm::vec4 v_colors[])`: Creates a quadrilateral from four vertices.
    - `void colorcube(double xmin, double xmax, double ymin, double ymax, std::string part)`: Generates a color cube, which is a building block for the model.
    - `void drawBody(void)`, `void drawCar(void)`: Draws specific parts of the transformer.
    - `void initBuffersGL(void)`, `void renderGL(void)`: Initialize buffers and handle rendering in OpenGL.
2. **OpenGL Framework (`gl_framework.cpp` & `gl_framework.hpp`):**
    - `void initGL(void)`: Initializes OpenGL settings.
    - `void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)`: Handles keyboard inputs, which might control the animation or interactions.
3. **Hierarchy Node Management (`hierarchy_node.cpp` & `hierarchy_node.hpp`):**
    - `void HNode::update_matrices()`: Updates the transformation matrices for the nodes in the hierarchy.

- `void HNode::add_child(HNode* a_child)`: Adds a child node to the current node.
- `void HNode::change_parameters(GLfloat atx, GLfloat aty, GLfloat atz, GLfloat arx, GLfloat ary, GLfloat arz)`: Changes the transformation parameters for a node.
- `void HNode::render()`, `void HNode::render_tree()`: Responsible for rendering the node and its children.

4. **Shader Utilities (`shader_util.cpp` & `shader_util.hpp`):**
    - Contains utilities to load and compile shaders, and handle any GLSL related errors.

## 2. System Architecture

### 2.1 Hierarchical Modeling

Hierarchical modeling is a cornerstone of the "Transformer Animation" project. This approach allows complex objects to be constructed from simpler components, with transformations applied hierarchically. The system treats each part of the transformer as a node in a tree structure, where the root node represents the entire model, and the child nodes represent individual parts such as arms, legs, or the body.

Each node maintains its own transformation matrices, which dictate its position, rotation, and scale relative to its parent node. This hierarchical structure is crucial for accurately animating the transformer, ensuring that movements like the rotation of an arm or leg are propagated correctly throughout the model.

### 2.2 Node Management

The `HNode` class is responsible for managing these nodes. The key functions within this class include:

- `update_matrices()`: Updates the transformation matrices for each node, ensuring that all transformations are correctly applied.
- `add_child(HNode* a_child)`: Adds a child node to the current node, allowing for complex hierarchies to be created.

- `change_parameters(GLfloat atx, GLfloat aty, GLfloat atz, GLfloat arx, GLfloat ary, GLfloat arz)`: Changes the transformation parameters (translation and rotation) for the node.
- `render()` and `render_tree()`: Responsible for rendering the node and recursively rendering all its children.

## 3. Shader Programs

### 3.1 Vertex and Fragment Shaders

The project employs two key shader programs: the vertex shader and the fragment shader. These shaders are written in GLSL and are responsible for determining how the geometry of the model is transformed and how it is shaded.

- **Vertex Shader (`07_vshader.glsl`)**: This shader handles the transformation of vertices from object space to screen space. It applies the model, view, and projection matrices to each vertex, which are essential for positioning the transformer in the 3D scene.
- **Fragment Shader (`07_fshader.glsl`)**: This shader determines the color of each pixel on the screen. It applies lighting calculations, which include ambient, diffuse, and specular components, to simulate realistic lighting effects on the transformer's surface.

The shaders are compiled and linked into a shader program using utilities in `shader_util.cpp`. Any errors encountered during this process are handled gracefully, ensuring that the program can provide informative feedback for debugging.

## 4. Key Functions

### 4.1 Hierarchical Modeling Functions

The hierarchical modeling is implemented through several critical functions in `07_hierarchical_modelling.cpp`:

- **`void setPositions(double xmin, double xmax, double ymin, double ymax)`**: Sets the positions for the vertices of the model. This function is crucial for defining the spatial boundaries of each part of the transformer.
- **`void quad(int a, int b, int c, int d, glm::vec4 v_positions[], glm::vec4 v_colors[])`**: Creates a quadrilateral from four vertices. This function is used to construct the basic geometry of the transformer.
- **`void colorcube(double xmin, double xmax, double ymin, double ymax, std::string part)`**: Generates a color cube that forms the building blocks of the transformer model. Each part of the transformer is constructed by combining multiple cubes.
- **`void drawBody(void)` and `void drawCar(void)`**: These functions draw specific parts of the transformer, such as the body and the vehicle form. They are called during the rendering process to construct the complete model.

### 4.2 OpenGL Framework Functions

The OpenGL framework is initialized and managed through the following functions in `gl_framework.cpp`:

- **`void initGL(void)`**: Initializes the OpenGL context and sets up the necessary states for rendering. This function is called at the start of the program to ensure that everything is set up correctly before rendering begins.
- **`void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)`**: Handles keyboard inputs, which allow the user to interact with the animation, such as triggering specific transformations or changing views.

## 5. Implementation Details

### 5.1 Buffer Initialization and Rendering

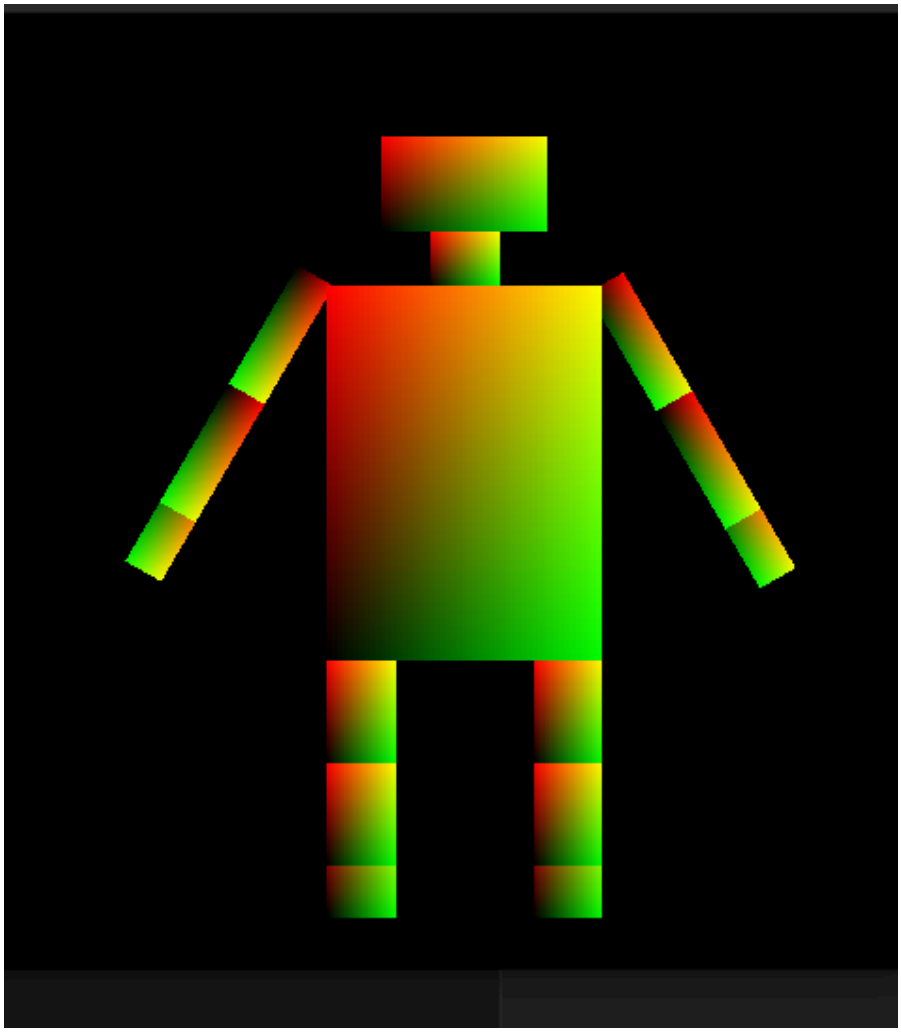
The project uses OpenGL buffers to store vertex data, colors, and indices, which are essential for rendering the transformer model efficiently. The `initBuffersGL(void)` function in `07_hierarchical_modelling.cpp` sets up these buffers, binding vertex data and initializing the VAO (Vertex Array Object) for the model.

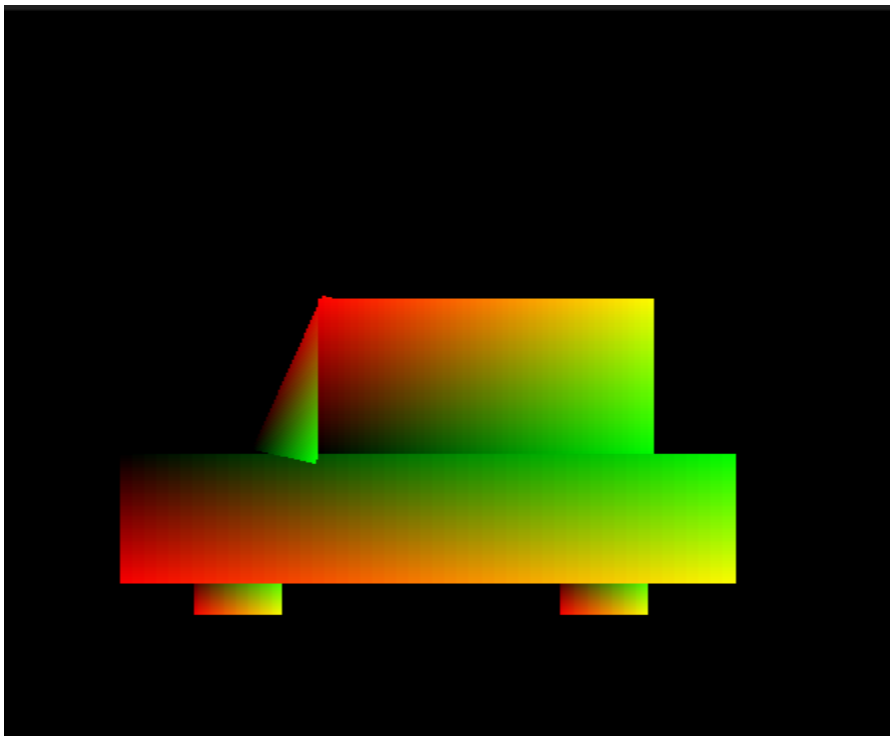
Rendering is handled by the `renderGL(void)` function, which is responsible for drawing the entire scene. This function calls the hierarchical node rendering functions, ensuring that each part of the transformer is drawn in the correct order and with the correct transformations applied.

### 5.2 Shader Compilation and Linking

Shader programs are compiled and linked using utility functions in `shader_util.cpp`. These functions load the shader source code from files, compile them, and link them into a complete program. The process is robust, with error handling mechanisms to catch and report any issues during compilation or linking.

## Final Output





## 6. Conclusion

The "Transformer Animation" project successfully demonstrates the power of hierarchical modeling in creating complex, articulated models in computer graphics. By combining sophisticated shader programs with a well-structured hierarchical system, the project achieves a realistic and dynamic animation of a transformer. The use of OpenGL and GLSL enables real-time rendering, making the animation interactive and visually compelling.

Despite the challenges encountered during development, such as managing the complexity of the hierarchical system and optimizing the rendering process, the project achieves its objectives. Future work could involve adding more sophisticated animations, improving the shading techniques, or expanding the model to include more detailed parts.

The "Transformer Animation" project stands as a testament to the possibilities of modern computer graphics, providing a foundation for further exploration and innovation in the field.