HW2 Report

110550110 林書愷

Main cpp:

Create shader:

```
unsigned int createShader(const string& filename, const string& type)
{
    unsigned int shader;
    if (type == "vert")
        shader = glCreateShader(GL_VERTEX_SHADER);
    else if (type == "frag")
        shader = glCreateShader(GL_FRAGMENT_SHADER);
    else
    {
        cout << "Unknown Shader Type.\n";
        return 0;
    }
```

```
    // Read the code of shader file
    ifstream fs(filename);
    stringstream ss;
    string s;
    while (getline(fs, s))
    {
        ss << s << "\n";
    }
    string temp = ss.str();
    const char* source = temp.c_str();

    // Compile shader
    glShaderSource(shader, 1, &source, NULL);
    glCompileShader(shader);
```

```
    // Debug info
    int success;
    char infoLog[512];
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(shader, 512, NULL, infoLog);
        cout << "ERROR::SHADER::" << type << "::COMPLIATION_FAILED\n" << infoLog << "\n";
        return 0;
    }

    return shader;
```

This function aims to create shader objects. In this assignment, there are only

two objects: vertex and fragment shader, so we only need to input two strings two decide which shader is creating in the session. The filename string will input a path of current shader's file path to that the function able to have access to shader creating source and the type string is used to decide what type of shader is going to create. As long as there are only two types, an if-else is good enough to judge what is going to be create and the condition will conduct a string compare to figure what to create now.

Then moving on to the next part is handling the filename path. Since input of gl functions are const char array, we need to convert the input string into const char array. To conduct this, implementing stringstream will be an easy way. First use ifstream to read in the path string and store it into a stringstream then convert it into string and finally use c_string function to convert it into a const char array.

After all pre-processing steps are done, we can finally build up shader. With the input path ready, we can now input it to the glshadersource function to set the source code for a shader object. The parameters are one string and one uint to memorize the shader object id. Then pass the shader uint to glcompileshader to compile the shader code.

Then moving on to debug session. We first set up an int success to store the state of the shader code. Using glgetshaderiv to check if the compilation was successful by querying the compilation status and store it into success. If the int is 1 means it failed so we move on the get the debug info with glgetshaderinfolog. Finally return the shader uint to finish shader create process.

Createprogram:

```
unsigned int createProgram(unsigned int vertexShader, unsigned int fragmentShader)
{
    unsigned int program = glCreateProgram();
    glAttachShader(program, vertexShader);
    glAttachShader(program, fragmentShader);
    glLinkProgram(program);
    glDetachShader(program, vertexShader);
    glDetachShader(program, fragmentShader);
    return program;
}
```

This function aims to create program objects. First use glcreateProgram to create an empty gl program that can be used to link to other shader program and the return int is the id of the created program. Then use glattachshader to attach shaders to the created, and since there are only two shaders: vertex and fragment in this assignment so just need to call this function twice. After the shaders are linked to the

program, we need to link the program to combine shader and create an executable program that can be used for rendering. This task will be done by gllinkprogram function. Finally after linking the program all we need for rendering is done, we no longer need the shader objects in our program so we detach it from our program to save memory. This task will be conducted with gldetachshader and using it twice since we got two shader objects to remove. After all jobs are done, return the id of program for further rendering usage.

Loadtexture:

```
unsigned int loadTexture(const char* tFileName) {
    unsigned int texture;
    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glActiveTexture(GL_TEXTURE0);
    int width, height, numChannels;
    stbi_set_flip_vertically_on_load(true);
    unsigned char* data = stbi_load(tFileName, &width, &height, &numChannels, 0);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
    glBindTexture(GL_TEXTURE_2D, 0);
    stbi_image_free(data);
    return texture;
}
```

This function aims to load an image file and create texture for rendering. Therefore the input definitely includes a string path of target image that is used for loading texture. First step of loading texture will be setting up texture types, call glenable to enable specific opengl feature. In this function we want to load texture from an image so we enable GL_Texture_2d to enable 2d texturing. Then calling glGentexture to set up an empty texture code for later usage and an uint is also needed to store the id of texture code. After the texture is generated, bind the 2d texture abd the texture code together with glbindtexture function. With this operation means that any subsequent operations that affect textures, such as setting parameters or loading data, will apply to the texture with the ID stored in the variable texture. As the binding step is done, we can begin to set the texture parameters for the gl_texture_min_filter it determines how the texture is sampled when its size on screen is smaller than the original size. GL_LINEAR specifies linear interpolation, providing smoother transitions between texels and mag_fiter is for sample that is larger than screen how it should acts. After all settings are done we active the texture code with glactivetexture function and as we only get one texture code we input gltexture0 for uint0 inputs.

With all texture pre-settings are done, we can know input our texture source into the texture code. To get the detail information of texture source including width,

height and number of channels needs to use stbi_load function to load the image from the file and returns char array to the image data. Noticely, before loading the image data need to use stbi_set_flip_vertically_on_load(true) as it sets a flag in the stb_image library to flip the image vertically during loading. OpenGL typically expects image data where the origin (0,0) is at the bottom-left, while many image formats store the origin at the top-left.

Once the image data is ready we can use glteximage2d to create the texture we want in texture code. Inputting the image data and set the internal format to gl_rgb to regulate the format into rgb style. Also need to input GL_TEXTURE_2D as the target texture type and the image texture is done.

Finally need to debind the texture code to prevent further modification while operating it so we call glbind texture again but this time we bind it with 2dtexture with null pointer. Also need to free image data array to avoid memory waste. In the end return the texture code id for later usage.

ModelVAO:

```
unsigned int modelVAO(Object& model)
{
    unsigned int VAO, VBO[3];
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);
    glGenBuffers(3, VBO);
```

```
glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (model.positions.size()), &(model.positions[0]), GL_STATIC_DRAW
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 3, 0);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (model.normals.size()), &(model.normals[0]), GL_STATIC_DRAW);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 3, 0);
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (model.texcoords.size()), &(model.texcoords[0]), GL_STATIC_DRAW
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 2, 0);
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindVertexArray(0);

return VAO;
```

Step 1: Generate Vertex Array Object (VAO) glGenVertexArrays(1, &VAO); glBindVertexArray(VAO); glGenVertexArrays(1,

&VAO); This generates a single Vertex Array Object (VAO) and stores its ID in the variable VAO.
glBindVertexArray(VAO);: This binds the VAO, making it the active VAO. Subsequent OpenGL commands related to vertex attributes and VBOs will be associated with this VAO.
Step 2: Generate and bind Vertex Buffer Objects (VBOs)
glGenBuffers(3, VBO),glGenBuffers(3, VBO); This generates three Vertex Buffer Objects (VBOs) for positions, normals, and texture coordinates. The IDs are stored in the array VBO.
Step 3: Bind and fill the first VBO with vertex positions
glBindBuffer(GL_ARRAY_BUFFER, VBO[0]): This binds the first VBO (for vertex positions) to the GL_ARRAY_BUFFER target. Subsequent operations will affect this buffer.
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * (model.positions.size()), &(model.positions[0]), GL_STATIC_DRAW): This allocates and fills the VBO with the vertex position data from the model.positions vector. The GL_STATIC_DRAW hint indicates that the data is not expected to change frequently.
Step 4: Bind and fill the second VBO with vertex
glBindBuffer(GL_ARRAY_BUFFER, VBO[1]): This binds the second VBO (for vertex normals) to the GL_ARRAY_BUFFER target.
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * (model.normals.size()), &(model.normals[0]), GL_STATIC_DRAW): This allocates and fills the VBO with the vertex normal data from the model.normals vector.
Step 5: Bind and fill the third VBO with texture coordinates
glBindBuffer(GL_ARRAY_BUFFER, VBO[2]): This binds the third VBO (for texture coordinates) to the GL_ARRAY_BUFFER target.
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * (model.texcoords.size()), &(model.texcoords[0]), GL_STATIC_DRAW): This allocates and fills the VBO with the texture coordinate data from the model.texcoords vector.
Step 6: Unbind VAO
glBindVertexArray(0);: This unbinds the currently bound VAO. It's a good practice to unbind the VAO after setting it up to avoid accidentally modifying it later.
Step 7: Return the VAO
Finally returns the ID of the VAO, which can be used later for rendering the model.
In summary, this function is responsible for creating a VAO and associated

VBOs for a 3D model. It binds and fills the VBOs with the model's vertex positions, normals, and texture coordinates. The resulting VAO is a container that stores the configuration needed to render the model efficiently.

Drawmodel:

```
void drawModel(const string& target, unsigned int& shaderProgram,
    const glm::mat4& M, const glm::mat4& V, const glm::mat4& P)
{
    unsigned int mLoc, vLoc, pLoc;
    mLoc = glGetUniformLocation(shaderProgram, "M");
    vLoc = glGetUniformLocation(shaderProgram, "V");
    pLoc = glGetUniformLocation(shaderProgram, "P");
    glUniformMatrix4fv(mLoc, 1, GL_FALSE, glm::value_ptr(M));
    glUniformMatrix4fv(vLoc, 1, GL_FALSE, glm::value_ptr(V));
    glUniformMatrix4fv(pLoc, 1, GL_FALSE, glm::value_ptr(P));
    if (target == "board") {
        glBindVertexArray(boardVAO);
        glDrawArrays(GL_TRIANGLES, 0, boardModel.positions.size());
    }
    else {
        glBindVertexArray(penguinVAO);
        glDrawArrays(GL_TRIANGLES, 0, penguinModel.positions.size());
    }

    glBindVertexArray(0);
}
```

This function is used for drawing out the generated model based on shader program, so the input will be string to decide what object is drawing now and shader program id , model, projection, view matrix. Uniform Locations Setup:
mLoc, vLoc, and pLoc are used to store the uniform locations for the model matrix (M), view matrix (V), and projection matrix (P) in the shader, respectively.
Step1. Setting Matrices in the Shader:
glUniformMatrix4fv is used to set the values of the model, view, and projection matrices in the shader using their respective uniform locations.
Step2. Drawing the Model:
The function checks the target parameter to determine which model to draw. If target is "board," it binds the VAO associated with the boardModel and draws it using glDrawArrays. If the target is not "board," it assumes the target is the "penguin" and draws the penguinModel.
The number of vertices to draw is determined by the size of the positions vector in the respective model.
Step3. Unbinding the VAO:
After drawing, it's a good practice to unbind the VAO to avoid unintentional modifications. glBindVertexArray(0) is used for this purpose.

Keycallback:

```
void keyCallback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
        cout << "KEY S PRESSED" << endl;
        if (squeezing) {
            squeezing = false;
        }
        else {
            squeezing = true;
        }
    }
```

```
    if (glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS) {
        cout << "KEY G PRESSED" << endl;
        if (useGrayscale) {
            useGrayscale = false;
        }
        else {
            useGrayscale = true;
        }
    }
    if (glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS) {
        cout << "KEY R PRESSED" << endl;
        if (rainbow) {
            rainbow = false;
        }
        else {
            rainbow = true;
        }
    }
```

```
    if (glfwGetKey(window, GLFW_KEY_E) == GLFW_PRESS) {
        cout << "KEY E PRESSED" << endl;
        transitionspeed /= 2;
    }
    if (glfwGetKey(window, GLFW_KEY_T) == GLFW_PRESS) {
        cout << "KEY T PRESSED" << endl;
        transitionspeed *= 2;
    }
```

This function is used for getting keyboard input and decide what value modification should be done. If the key s is pressed then it will modify the squeezing bool variable to its compliment value. If key g is pressed, then it will modify the useGrayscale bool variable to its compliment value too. Key r also modify a bool variable rainbow, this is for the bonus part to whether it start or not. These three bool values will be later passed to shaders to control what segment of code should be executed. For the key e pressed, it will modify the int transitionspeed which will be times to the currenttime variable and passed to the fragment shader to speed up color changing, and e will let the speed divde with 2 while T will let it times 2.

Display loop:

Shader and program creating:

```
unsigned int vertexShader, fragmentShader, shaderProgram;
vertexShader = createShader("vertexShader.vert", "vert");
fragmentShader = createShader("fragmentShader.frag", "frag");
shaderProgram = createProgram(vertexShader, fragmentShader);
glUseProgram(shaderProgram);
```

Setting up vertex and fragment shader by passing the shader source path and its type string and memorize the setup function's return shader code id. After two shaders are done pass it to createProgram function to create program use the return program id to activates the shader program with gluseprogram which let the subsequent rendering commands will use the shaders specified in this program.

Loading texture:

```
unsigned int penguinTexture, boardTexture;
penguinTexture = loadTexture("obj/penguin_diffuse.jpg");
boardTexture = loadTexture("obj/surfboard_diffuse.jpg");
```

This part will setup texture for two objects: penguin and board with passing there texture source file path into the loadTexture function.

Create VAO:

```
penguinVAO = modelVAO(penguinModel);
boardVAO = modelVAO(boardModel);
```

Call global variable obj penguinModel and boardModel into the modelVAO to create models for later rendering.

Getting location from shaderprogram:

```
GLint modelLocation = glGetUniformLocation(shaderProgram, "model");
GLint viewLocation = glGetUniformLocation(shaderProgram, "view");
GLint projectionLocation = glGetUniformLocation(shaderProgram, "projection");
GLint colorLocation = glGetUniformLocation(shaderProgram, "color");
GLint squeezeFactorLocation = glGetUniformLocation(shaderProgram, "squeezeFactor");
GLint grayscaleLocation = glGetUniformLocation(shaderProgram, "grayscale");
```

GLint modelLocation = glGetUniformLocation(shaderProgram, "model");
This line retrieves the location of the uniform variable named "model" in the shader program identified by shaderProgram.
modelLocation will hold the location of the "model" uniform.
GLint viewLocation = glGetUniformLocation(shaderProgram, "view");

Similar to the first line, but it retrieves the location of the uniform variable named "view."
GLint projectionLocation = glGetUniformLocation(shaderProgram, "projection");
Retrieves the location of the uniform variable named "projection."
GLint colorLocation = glGetUniformLocation(shaderProgram, "color");
Retrieves the location of the uniform variable named "color."
GLint squeezeFactorLocation = glGetUniformLocation(shaderProgram, "squeezeFactor");
Retrieves the location of the uniform variable named "squeezeFactor."
GLint grayscaleLocation = glGetUniformLocation(shaderProgram, "grayscale");
Retrieves the location of the uniform variable named "grayscale."

Creating board:

```
glm::mat4 board(1.0f);
board = glm::translate(board, glm::vec3(0.0f,-0.5f,swingPos));
board = glm::rotate(board, glm::radians(-swingAngle), glm::vec3(0.0f, 1.0f, 0.0f));
board = glm::rotate(board, glm::radians(-90.0f), glm::vec3(1.0f, 0.0f, 0.0f));
board = glm::scale(board, glm::vec3(0.03f, 0.03f, 0.03f));
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, boardTexture);
glUniform1i(glGetUniformLocation(shaderProgram, "useGrayscale"), useGrayscale);
glUniform1f(glGetUniformLocation(shaderProgram, "squeezeFactor"), 0);
glUniform1i(glGetUniformLocation(shaderProgram, "rainbow"), rainbow);
glUniform1f(glGetUniformLocation(shaderProgram, "hue"), hue);
drawModel("board", shaderProgram, board, view, perspective);
```

This part is drawing out the board model. First set the board matrix to required position ,angle and scale. Also the translate will be swingPos as it will surf and the rotate will have as one is for swingAngle for surfing and the input angle need to be negative as the model is respective to the camera we seen Then active the index 0 texture in the texture code with glActiveTexture function and bind the require boardTexture with glbindTexture function. After that pass in additive variables include useGrayscale(for fragment shader to judge gray scale mod is active or not) and squeezefactor pass in 0 since board don’t need to squeeze and rainbow to control rainbow mod is active or not and hue as float pass in to fragment shader to judge which color is now changing to. Finally use draw model function to draw them out.

Creating penguin:

```
glm::mat4 penguin(1.0f);
penguin = glm::translate(penguin, glm::vec3(0.0f, 0.0f, swingPos));
penguin = glm::rotate(penguin, glm::radians(-swingAngle), glm::vec3(0.0f, 1.0f,0.0f));
penguin = glm::rotate(penguin, glm::radians(-90.0f), glm::vec3(1.0f, 0.0f, 0.0f));

penguin = glm::scale(penguin, glm::vec3(0.025f, 0.025f, 0.025f));
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, penguinTexture);
glUniform1i(glGetUniformLocation(shaderProgram, "useGrayscale"), useGrayscale);
glUniform1f(glGetUniformLocation(shaderProgram, "squeezeFactor"), squeezeFactor);
drawModel("penguin", shaderProgram, penguin, view, perspective);
```

This part is drawing out the penguin model. The steps are nearly same as board but this part need to passing squeezefactor instead of zero since it have the squeezing mod.

Additive function settings:

```
if (swingway) {
    swingAngle += 20.0 * dt;
    if (swingAngle >= 20.0) {
        swingway = false;
    }
}
else {
    swingAngle -= 20.0 * dt;
    if (swingAngle <= -20.0) {
        swingway = true;
    }
}
```

This is for the swingangle modify. I set up another bool variable swingway to judge what is the current swinging way if its true then swingAngle will be add 20 * dt every loop iteration. Once it exceed 20 it will change swingway value and move as another way. The other way is same as positive one but its limitation is -20.

```
if (moveway) {
    swingPos += 1.0 * dt;
    if (swingPos >= 2.0) {
        //swingPos = 2.0;
        moveway = false;
    }
}
else {
    swingPos -= 1.0 * dt;
    if (swingPos <= 0.0) {
        //swingPos = 0.0;
        moveway = true;
    }
}
```

This if-else is for the surfing move way. Again I set up another bool variable moveway to judge current moving way. The mechanism is similar to swing but the difference is it add 1.0*dt to the swingPos and the limitation is 2.0 to 0.0.

```
if(squeezing){
    squeezeFactor += 90.0*dt;
}
```

This one is for squeezingfactor accumulation. It will be active once the squeezing mode is on and add 90*dt every iteration.

```
if (rainbow) {
    hue+= transitionspeed * dt;
}
```

This is the bonus part for the rainbow color mode. It will accumulate current color changing amount by adding transitionspeed * dt. It will be active if the rainbow mode is on and the hue is a float variable that will be passed to fragment shader.

Vertex shader:

```
vec3 originalPos = aPos;
vec3 squeezedPos;
float newY = aPos.y + aPos.z * sin(radians(squeezeFactor)) / 2.0;
float newZ = aPos.z + aPos.y * sin(radians(squeezeFactor)) / 2.0;
squeezedPos = vec3(aPos.x, newY, newZ);
vec4 worldPos = M * vec4(squeezedPos, 1.0);
gl_Position = P * V * worldPos;

mat3 normalMatrix = transpose(inverse(mat3(M)));
normal = normalize(normalMatrix * aNormal);

// Pass the texture coordinates to the fragment shader
texCoord = aTexCoord;
```

This is the vertex shader main part and there is no new set up variable. First I get a copy of aPos to prevent the value been contaminate by calculation. The calculate the new y and z by the give formula and use the passed in squeezeFactor. Then merge the original x and new y z value into a vec3 and multiple with the passed in model, projection and view matrix.

Then do the normalization with inversing the model matrix and transpose it and finally times the last iteration's normal matrix to normalize.

Finally set the texture coordinates to fragment shader to finish it.

Fragment shader:

```
uniform bool rainbow;

uniform float hue;
```

These are the additional variables for bonus part.

```
vec3 hsv2rgb(vec3 c) {
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}
```

This function is for converting a color from the HSV (Hue, Saturation, Value) color space to the RGB (Red, Green, Blue) color space. The mechanism

is c is a vec3 representing the input color in HSV space, where c.x is the hue, c.y is the saturation, and c.z is the value.
K is a vec4 containing constants used in the conversion.
fract(c.xxx + K.xyz) calculates the fractional part of (c.x + K.x, c.x + K.y, c.x + K.z). This is done to ensure that the hue value is within the range [0, 1).
abs(...) takes the absolute value of the result.
* 6.0 - K.www scales the values by 6 and subtracts the constants from K.www.
clamp(p - K.xxx, 0.0, 1.0) ensures that the resulting values are within the range [0, 1].
mix(K.xxx, ..., c.y) linearly interpolates between K.xxx and the clamped values based on the saturation (c.y).
c.z * ... scales the result by the value (c.z).

```
vec4 color = texture(ourTexture, texCoord);
if (useGrayscale) {
    float grayscaleValue = dot(color.rgb, vec3(0.299, 0.587, 0.114));
    FragColor = vec4(grayscaleValue, grayscaleValue, grayscaleValue, color.a);
}
else if(rainbow){
    vec3 rgb = hsv2rgb(vec3(mod(hue, 6.0), 1.0, 1.0));
    FragColor = vec4(rgb, 1.0);
}
else {
    FragColor = color;
}
```

This is the main part of fragment shader and it has three mode: grayscale rainbow and normal:

vec4 color = texture(ourTexture, texCoord);: Fetches the color from a texture (ourTexture) at the specified texture coordinates (texCoord).
Grayscale Conversion (if useGrayscale is true): Calculates the grayscale value by taking the dot product of the color's RGB components and a vector representing the luminance (brightness) values of red, green, and blue.
Assigns a grayscale color with the alpha channel unchanged.
Rainbow Coloring (if rainbow is true): Converts the color to a rainbow color using the hsv2rgb function. The hue (hue) is modulated to ensure it stays within the valid HSV range.
Assigns the resulting RGB color with an alpha value of 1.0.
No Modification (if neither useGrayscale nor rainbow is true): If neither useGrayscale nor rainbow is true, the original color is used without modification.

Problem & bonus:

The biggest problem I met is in the bonus part. In the bonus part I want

to make my model able to change with every iteration of the while loop. Yet in the beginning I try to modify the color vec4 every second with a specific value instead of a formula. The outcome showed to be very unnatural and wired. So I try to use the hsv2rgb way to change the color and the outcome improved significantly as it change the color smoothly.(the solve function is the one mentioned above)