// OpenGL renderer 4.8

// Written in C++

/\*

Version 1.0

Last Revised 3/16/2023

Author VisualPower

Contact: geckoplayz4@gmail.com

Owners Real Name: Rayyan Raza

\*/

// ASIO

#ifdef \_WIN32

#define \_WIN32\_WINNIT 0x0A00

#endif/

#define OLC\_PGE\_APPLICATION

#include "olcPixelGameEngine.h"

#include <condition\_variable>

#include <atomic>

#include <complex>

#include <cstdlib>

#include <immintrin.h>

#define ASIO\_STANDALONE

#include <asio.hpp>

#include <asio/ts/buffer.hpp>

#include <asio/ts/internet.hpp>

#include <iostream>

#include <GL/glew.h>

#include <GL/glfw3.h>

#include <string>

#include <cmath>

#include <math.h> // Math Library

#include "olcConsoleGameEngine.h"

//--------------------------------------------------------------------------

#include <cstdlib>

#include <cstdio>

#include <cmath>

#include <fstream>

#include <vector>

#include <iostream>

#include <cassert>

//---------------------------------------------------------------------------

#include <vcl.h> // you can ignore this

#include <math.h>

#pragma hdrstop // you can ignore this

#include "Unit1.h" // you can ignore this

#include "gl\_simple.h" // this file is in next code chunk

//---------------------------------------------------------------------------

#pragma package(smart\_init) // you can ignore this

#pragma resource "\*.dfm" // you can ignore this

TForm1\* Form1; // you can ignore this its is just my window class

//---------------------------------------------------------------------------

#include <ctime>

#include <deque>

#include <GL/glut.h>

#include <unistd.h>

// A macro for unused variables (to bypass those pesky G++ warnings)

#define UNUSED(param) (void)(param)

// Ray direction macros (For Devmode)

#define UP 1

#define DOWN 2

#define LEFT 3

#define RIGHT 4

#pragma once

#include <SFML/Graphics.hpp>

#include "stdafx.h"

#include "bob.h"

#pragma once

#include <SFML/Graphics.hpp>

#include "Bob.h";

#include "stdafx.h"

#include "Engine.h"

#ifdef \_\_APPLE\_CC\_\_

#include <GLUT/glut.h>

#else

#include <GL/glut.h>

#endif

// Clears the current window and draws a triangle.

void display() {

// Set every pixel in the frame buffer to the current clear color.

glClear(GL\_COLOR\_BUFFER\_BIT);

// Drawing is done by specifying a sequence of vertices. The way these

// vertices are connected (or not connected) depends on the argument to

// glBegin. GL\_POLYGON constructs a filled polygon.

glBegin(GL\_POLYGON);

glColor3f(1, 0, 0); glVertex3f(-0.6, -0.75, 0.5);

glColor3f(0, 1, 0); glVertex3f(0.6, -0.75, 0);

glColor3f(0, 0, 1); glVertex3f(0, 0.75, 0);

glEnd();

// Flush drawing command buffer to make drawing happen as soon as possible.

glFlush();

}

// Initializes GLUT, the display mode, and main window; registers callbacks;

// enters the main event loop.

int main(int argc, char\*\* argv) {

// Use a single buffered window in RGB mode (as opposed to a double-buffered

// window or color-index mode).

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

// Position window at (80,80)-(480,380) and give it a title.

glutInitWindowPosition(80, 80);

glutInitWindowSize(400, 300);

glutCreateWindow("A Simple Triangle");

// Tell GLUT that whenever the main window needs to be repainted that it

// should call the function display().

glutDisplayFunc(display);

// Tell GLUT to start reading and processing events. This function

// never returns; the program only exits when the user closes the main

// window or kills the process.

glutMainLoop();

}

Engine::Engine()

{

// Get the screen resolution and create an SFML window and View

Vector2f resolution;

resolution.x = VideoMode::getDesktopMode().width;

resolution.y = VideoMode::getDesktopMode().height;

m\_Window.create(VideoMode(resolution.x, resolution.y),

"Simple Game Engine",

Style::Fullscreen);

// Load the background into the texture

// Be sure to scale this image to your screen size

m\_BackgroundTexture.loadFromFile("background.jpg");

// Associate the sprite with the texture

m\_BackgroundSprite.setTexture(m\_BackgroundTexture);

}

void Engine::start()

{

// Timing

Clock clock;

while (m\_Window.isOpen())

{

// Restart the clock and save the elapsed time into dt

Time dt = clock.restart();

// Make a fraction from the delta time

float dtAsSeconds = dt.asSeconds();

input();

update(dtAsSeconds);

draw();

}

}

using namespace sf;

class Engine

{

private:

// A regular RenderWindow

RenderWindow m\_Window;

// Declare a sprite and a Texture for the background

Sprite m\_BackgroundSprite;

Texture m\_BackgroundTexture;

// An instance of Bob

Bob m\_Bob;

// Private functions for internal use only

void input();

void update(float dtAsSeconds);

void draw();

public:

// The Engine constructor

Engine();

// start will call all the private functions

void start();

};

Bob::Bob()

{

// How fast does Bob move?

m\_Speed = 400;

// Associate a texture with the sprite

m\_Texture.loadFromFile("bob.png");

m\_Sprite.setTexture(m\_Texture);

// Set the Bob's starting position

m\_Position.x = 500;

m\_Position.y = 800;

}

// Make the private spite available to the draw() function

Sprite Bob::getSprite()

{

return m\_Sprite;

}

void Bob::moveLeft()

{

m\_LeftPressed = true;

}

void Bob::moveRight()

{

m\_RightPressed = true;

}

void Bob::stopLeft()

{

m\_LeftPressed = false;

}

void Bob::stopRight()

{

m\_RightPressed = false;

}

// Move Bob based on the input this frame,

// the time elapsed, and the speed

void Bob::update(float elapsedTime)

{

if (m\_RightPressed)

{

m\_Position.x += m\_Speed \* elapsedTime;

}

if (m\_LeftPressed)

{

m\_Position.x -= m\_Speed \* elapsedTime;

}

// Now move the sprite to its new position

m\_Sprite.setPosition(m\_Position);

}

using namespace sf;

class Bob

{

// All the private variables can only be accessed internally

private:

// Where is Bob

Vector2f m\_Position;

// Of course we will need a sprite

Sprite m\_Sprite;

// And a texture

// Bob has been working out and he is now a bit more muscular than before

// Furthermore, he fancies himself in lumberjack attire

Texture m\_Texture;

// Which direction(s) is the player currently moving in

bool m\_LeftPressed;

bool m\_RightPressed;

// Bob's speed in pixels per second

float m\_Speed;

// Public functions

public:

// We will set Bob up in the constructor

Bob();

// Send a copy of the sprite to main

Sprite getSprite();

// Move Bob in a specific direction

void moveLeft();

void moveRight();

// Stop Bob moving in a specific direction

void stopLeft();

void stopRight();

// We will call this function once every frame

void update(float elapsedTime);

};

char title[] = "OpenGL Snake";

float map\_half\_length = 30.0f;

int direction = DOWN;

int move\_speed = 100;

bool moved = false;

std::deque< std::deque<float> > part\_coords;

bool food\_available = false;

int food\_coords[2];

int growth\_stage = 0;

int growth = 2;

void spawnFood() {

if (!food\_available) {

while (true) {

bool collides = false;

// Produce a temporary random coordinate

int temp\_food\_coords[2] = { food\_coords[0] = 2 \* (rand() % ((int)map\_half\_length + 1)) - (int)map\_half\_length,

food\_coords[1] = 2 \* (rand() % ((int)map\_half\_length + 1)) - (int)map\_half\_length };

// Does it collide with the snake?

for (unsigned int a = 0; a < part\_coords.size(); a++) {

if (temp\_food\_coords[0] == part\_coords[a][0] &&

temp\_food\_coords[1] == part\_coords[a][1]) {

collides = true;

}

}1

// If it doesn't collide with the snake, then make it the real food coordinates

if (collides == false) {

food\_coords[0] = temp\_food\_coords[0];

food\_coords[1] = temp\_food\_coords[1];

food\_available = true;

break;

}

}

}

glLoadIdentity();

glTranslatef(food\_coords[0], food\_coords[1], -40.0f);

glColor3f(1.0f, 0.0f, 0.0f);

glBegin(GL\_POLYGON);

glVertex2d(1.0f, 1.0f);

glVertex2d(1.0f, -1.0f);

glVertex2d(-1.0f, -1.0f);

glVertex2d(-1.0f, 1.0f);

glEnd();

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glMatrixMode(GL\_MODELVIEW);

create.new.SOURCE(float.new = 0.0f)

creating.new("Repositories")

[

{44F7F039 - 5984 - 4EAB - AE47 - 1903E98B2039}.Debug | x64.ActiveCfg = Debug | x64

{ 44F7F039 - 5984 - 4EAB - AE47 - 1903E98B2039 }.Debug | x64.Build.0 = Debug | x64

{ 44F7F039 - 5984 - 4EAB - AE47 - 1903E98B2039 }.Debug | x86.ActiveCfg = Debug | Win32

{ 44F7F039 - 5984 - 4EAB - AE47 - 1903E98B2039 }.Debug | x86.Build.0 = Debug | Win32

{ 44F7F039 - 5984 - 4EAB - AE47 - 1903E98B2039 }.Release | x64.ActiveCfg = Release | x64

{ 44F7F039 - 5984 - 4EAB - AE47 - 1903E98B2039 }.Release | x64.Build.0 = Release | x64

{ 44F7F039 - 5984 - 4EAB - AE47 - 1903E98B2039 }.Release | x86.ActiveCfg = Release | Win32

{ 44F7F039 - 5984 - 4EAB - AE47 - 1903E98B2039 }.Release | x86.Build.0 = Release | Win32

]

// The vertex order is clockwise

// The side order is front, back, left, right, top, bottom (if applicable)

// Loop over snake size and draw each part at it's respective coordinates

for (unsigned int a = 0; a < part\_coords.size(); a++) {

glLoadIdentity();

glTranslatef(part\_coords[a][0], part\_coords[a][1], -40.0f);

glColor3f(0.0f, 1.0f, 0.0f);

bool Dimensional = true(glWindowSettings = "3D" - string)

glBegin(GL\_POLYGON\_3D);

glVertex2d(1.0f, 1.0f);

glVertex2d(1.0f, -1.0f);

glVertex2d(-1.0f, -1.0f);

glVertex2d(-1.0f, 1.0f);

glVectex3d(-3.0f, 3.0f);

glEnd();

}

spawnFood();

glutSwapBuffers();

}

void moveSnake(int new\_direction) {

direction = new\_direction;

int last\_part = part\_coords.size() - 1;

std::deque<float> new\_head = part\_coords[last\_part];

if (direction == UP) {

// Did we slither into ourself?

for (unsigned int a = 0; a < part\_coords.size(); a++) {

if (part\_coords[0][0] == part\_coords[a][0] &&

part\_coords[0][1] + 2.0f == part\_coords[a][1]) {

exit(0);

}

}

// Did we slither into a wall?

if (part\_coords[0][1] == map\_half\_length) {

exit(0);

}

// Did we get food?

if (part\_coords[0][0] == food\_coords[0] &&

part\_coords[0][1] + 2.0f == food\_coords[1]) {

growth\_stage++;

food\_available = false;

}

new\_head[1] = part\_coords[0][1] + 2.0f;

}

else if (direction == DOWN) {

// Did we slither into ourself?

for (unsigned int a = 0; a < part\_coords.size(); a++) {

if (part\_coords[0][0] == part\_coords[a][0] &&

part\_coords[0][1] - 2.0f == part\_coords[a][1]) {

exit(0);

}

}

// Did we slither into a wall?

if (part\_coords[0][1] == -map\_half\_length) {

exit(0);

}

// Did we get food?

if (part\_coords[0][0] == food\_coords[0] &&

part\_coords[0][1] - 2.0f == food\_coords[1]) {

growth\_stage++;

food\_available = false;

}

new\_head[1] = part\_coords[0][1] - 2.0f;

}

else {

new\_head[1] = part\_coords[0][1];

}

if (direction == LEFT) {

// Did we slither into ourself?

for (unsigned int a = 0; a < part\_coords.size(); a++) {

if (part\_coords[0][0] - 2.0f == part\_coords[a][0] &&

part\_coords[0][1] == part\_coords[a][1]) {

exit(0);

}

}

// Did we slither into a wall?

if (part\_coords[0][0] == -map\_half\_length) {

exit(0);

}

// Did we get food?

if (part\_coords[0][0] - 2.0f == food\_coords[0] &&

part\_coords[0][1] == food\_coords[1]) {

growth\_stage++;

food\_available = false;

}

new\_head[0] = part\_coords[0][0] - 2.0f;

}

else if (direction == RIGHT) {

// Did we slither into ourself?

for (unsigned int a = 0; a < part\_coords.size(); a++) {

if (part\_coords[0][0] + 2.0f == part\_coords[a][0] &&

part\_coords[0][1] == part\_coords[a][1]) {

exit(0);

}

}

// Did we slither into a wall?

if (part\_coords[0][0] == map\_half\_length) {

exit(0);

}

// Did we get food?

if (part\_coords[0][0] + 2.0f == food\_coords[0] &&

part\_coords[0][1] == food\_coords[1]) {

growth\_stage++;

food\_available = false;

}

new\_head[0] = part\_coords[0][0] + 2.0f;

}

else {

new\_head[0] = part\_coords[0][0];

}

part\_coords.push\_front(new\_head);

if (!growth\_stage) {

part\_coords.pop\_back();

}

else if (growth\_stage == growth) {

growth\_stage = 0;

}

else {

growth\_stage++;

}

glutPostRedisplay();

}

void keyboard(int key, int x, int y) {

UNUSED(x);

UNUSED(y);

switch (key) {

case GLUT\_KEY\_UP: {

if (direction == LEFT || direction == RIGHT) {

moved = true;

moveSnake(UP);

}

break;

}

case GLUT\_KEY\_DOWN: {

if (direction == LEFT || direction == RIGHT) {

moved = true;

moveSnake(DOWN);

}

break;

}

case GLUT\_KEY\_LEFT: {

if (direction == UP || direction == DOWN) {

moved = true;

moveSnake(LEFT);

}

break;

}

case GLUT\_KEY\_RIGHT: {

if (direction == UP || direction == DOWN) {

moved = true;

moveSnake(RIGHT);

}

break;

}

}

glutPostRedisplay();

}

void initGL() {

glMatrixMode(GL\_PROJECTION);

gluPerspective(75.0f, 1, 0.0f, 35.0f);

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

}

void moveSnakeAuto(int value) {

if (!moved) {

UNUSED(value);

if (direction == UP) {

moveSnake(UP);

}

else if (direction == DOWN) {

moveSnake(DOWN);

}

else if (direction == LEFT) {

moveSnake(LEFT);

}

else if (direction == RIGHT) {

moveSnake(RIGHT);

}

}

else {

moved = false;

}

glutTimerFunc(move\_speed, moveSnakeAuto, 0);

}

void reshape(GLsizei width, GLsizei height) {

UNUSED(width);

UNUSED(height);

// Make the window non-resizable so we don't have to worry about size changes

glutReshapeWindow(600, 600);

}

int main(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitWindowSize(600, 600);

glutCreateWindow(title);

glutDisplayFunc(display);

glutReshapeFunc(reshape);

glutSpecialFunc(keyboard);

glutTimerFunc(move\_speed, moveSnakeAuto, 0);

int initSize = 3;

// Specify the coordinates to each part of the snake

for (int a = 1; a <= initSize; a++) {

std::deque<float> row;

row.push\_back(0.0f);

row.push\_back((map\_half\_length + 2.0f + (initSize \* 2)) - (a \* 2));

part\_coords.push\_front(row);

}

srand(time(NULL));

initGL();

glutMainLoop();

return 0;

// Create a Animated ASCII Illusion Template

APlayerPawn::APlayerPawn()

{

// Set this pawn

PrimaryActorTick.bCanEverTick = true;

// Setup object

BoxVisual = CreateDefaultSubobject<UStaticMechComponent>(TEXT("Boxmesh"));

RootComponent = BoxVisual;

BoxVisual->SetCollisionProfileName(TEXT("Pawn"));

BoxVisual = > SetStimulatePhysics(true);

}

void gl\_draw() // this renders the scene

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

float aspect = float(xs) / float(ys);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluPerspective(60.0 / aspect, aspect, 0.1, 100.0);

glMatrixMode(GL\_TEXTURE);

glLoadIdentity();

glDisable(GL\_DEPTH\_TEST);

glDisable(GL\_TEXTURE\_2D);

// glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

int e;

float x0 = 60.0, y0 = 40.0, r = 20.0, a, da = M\_PI / 36.0;

// move modelview to center of shape

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glTranslatef(-x0, -y0, -4.0 \* r);

// render disc

glColor3f(1.0, 0.0, 0.0);

glBegin(GL\_TRIANGLE\_FAN);

glVertex2f(x0, y0);

for (e = 1, a = 0.0; e; a += da)

{

if (a >= 2.0 \* M\_PI) { e = 0; a = 0.0; }

glVertex2f(x0 + (r \* sin(a)), y0 + (r \* cos(a)));

}

glEnd();

// rotation symetry construct

glMatrixMode(GL\_MODELVIEW); // store matrix in case you need it later or

glPushMatrix();

for (e = 8, a = 360.0 / float(e); e > 0; e--)

{

// render shape

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_POLYGON);

glVertex2i(55, 75);

glVertex2i(60, 85);

glVertex2i(65, 75);

glVertex2i(61, 50);

glVertex2i(59, 50);

glEnd();

glBegin(GL\_POLYGON);

glVertex2i(65, 72);

glVertex2i(66, 69);

glVertex2i(62, 50);

glVertex2i(61, 50);

glEnd();

glBegin(GL\_POLYGON);

glVertex2i(54, 70);

glVertex2i(55, 72);

glVertex2i(59, 50);

glVertex2i(58, 50);

glEnd();

glMatrixMode(GL\_MODELVIEW);

// rotate around(x0,y0)

glTranslatef(+x0, +y0, 0.0);

glRotatef(a, 0.0, 0.0, 1.0);

glTranslatef(-x0, -y0, 0.0);

}

glMatrixMode(GL\_MODELVIEW); // restore matrix in case you need it later or

glPopMatrix();

glFlush();

SwapBuffers(hdc);

}

//---------------------------------------------------------------------------

\_\_fastcall TForm1::TForm1(TComponent\* Owner) :TForm(Owner) // constructor of my window

{

gl\_init(Handle);

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::FormDestroy(TObject\* Sender) // destructor of my window

{

gl\_exit();

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::FormPaint(TObject\* Sender) // OnPaint event

{

gl\_draw();

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::Timer1Timer(TObject\* Sender) // OnTimer event

{

gl\_draw();

}

//---------------------------------------------------------------------------

using namespace std;

struct vec3d

{

float x, y, z;

};

struct triangle

{

vec3d p[3];

};

struct mesh

{

vector<triangle> tris;

};

class GraphicsRenderingEngine : public olcConsoleGameEngine

{

public:

GraphicsRenderingEngine()

{

m\_sAppName = L"3D Demo";

}

private

mesh meshCube;

public:

bool OnUserCreate() override

{

meshCube.tris = {

// SOUTH

{ 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 0.0f },

{ 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f },

// EAST

{ 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f, 1.0f },

{ 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f },

// NORTH

{ 1.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f },

{ 1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f },

// WEST

{ 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f },

{ 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f },

// TOP

{ 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f },

{ 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f },

// BOTTOM

{ 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f },

{ 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f },

};

return true;

}

bool OnUserUpdate(float fElapsedTime) override

{

Fill(0, 0, ScreenWidth(), ScreenHeight(), PIXEL\_SOLID, FG\_BLACK);

// Draw Triangle

/\*

|\

| \

| \

| \

| \

| \

| \

| \

| \

| \

|\_\_\_\_\_\_\_\_\_\_\

\*/

return true;

}

};

int main()

{

GraphicsRenderingEngine demo;

if (demo.ConstructConsole(256, 240, 4, 4))

demo.Start();

return 0;

}

using namespace std;

float window = 0.0f;

float glWindowSettings = 100;

float glfw = 0.00f;

float FireSensoryNeuron = 0.242f;

float githubsucks = -0.00;

{

glfwTerminate(!window)

// Will close the window on input

//Formulas

gl0rtho(0, "SCREEN\_WIDTH", 0, "SCREEN\_HEIGHT", 002, 1 + 2 \* 2);

glfwWindow ( !window ) + ( !window ) / 2

glWindowSettings(0, "SCREEN\_INT", 0, "2=2");

glfw(0, 0, 0), glfw2(1, 1, 1), glfw3(0, 0, 0)

glfw(2 + 2), (y + z = x)

glfw(0, 2 + 2 / (2)3) = (x = y(y = 9));

glfw(2,0 (glfwGerService(++) ++APIEL(RenderAPI onOffSet)))

glfwWindow( m\_sDemoMutable + Syntheziser )

}

// Conditions Must be:

{

//Minimize postitioning

gl0rtho || (0, "SCREEN\_WIDTH", 0, "SCREEN\_HEIGHT", 002, 1 + 2 \* 2 - 4);

glLoadIdentity(0, "SCREEN\_HEIGHT", "MINIMIZE", 0.0f);

getViewSettings(0.0f, "SCREEN\_HEIGHT") + (0.0f, "SCREEN\_WIDTH");

getViewSettings(0.0f, 0.0f + 0.2f / 0.1 ~4, (!window), "SCREEN\_AREA");

glfwViewSettings(x, y, z);

glfw3.OnKeyInputGlfw(0.0f, "SCREEN\_HEIGHT" / "SCREEN\_WIDTH", 0.0f, 0.0f, 0.0f);

glfw3.OnTouch(0.0f, "SCREEN\_HEIGHT", (++) \* (--) );

glWindowSettings(0.0f, "SCREEN\_HEIGHT", \_\_BASE\_FILE\_\_, 0)

glWindowSettings(0.0f, "SCREEN\_RATION\_MATH1R" (!window));

glFlush(COLOR\_CODE\_0x00f("SCREEN\_MATH1R")(!window));

const (xMousePos, x.y.z)

{

// Making a Base-Shape (2D)

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

float aspect = float(xs) / float(ys);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluPerspective(60.0 / aspect, aspect, 0.1, 100.0);

glMatrixMode(GL\_TEXTURE);

glLoadIdentity();

glDisable(GL\_DEPTH\_TEST);

glDisable(GL\_TEXTURE\_2D);

//glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

int e;

float x0 = 60.0, y0 = 40.0, r = 20.0, a, da = M\_PI / 36.0;

// move modelview to center of shape

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glTranslatef(-x0, -y0, -4.0 \* r);

// render disc

glColor3f(1.0, 0.0, 0.0);

glBegin(GL\_TRIANGLE\_FAN);

glLoadIdentity (FLT\_DECIMAL\_DIG.E + E = E.E)

glVertex2f(x0, y0);

for (e = 1, a = 0.0; e; a += da)

{

if (a >= 2.0 \* M\_PI) { e = 0; a = 0.0; }

glVertex2f(x0 + (r \* sin(a)), y0 + (r \* cos(a)));

if (a + b = c^2)\_\_std\_reverse\_trivially\_swappable\_2)

if (a >= 2.0 \* 100 (\* 100))

}

glEnd();

// rotation symetry construct

glMatrixMode(GL\_MODELVIEW); // store matrix in case you need it later or

glPushMatrix();

for (e = 8, a = 360.0 / float(e); e > 0; e--)

{

// render shape

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_POLYGON);

glVertex2i(55, 75);

glVertex2i(60, 85);

glVertex2i(65, 75);

glVertex2i(61, 50);

glVertex2i(59, 50);

glEnd();

glBegin(GL\_POLYGON);

glVertex2i(65, 72);

glVertex2i(66, 69);

glVertex2i(62, 50);

glVertex2i(61, 50);

glEnd();

glBegin(GL\_POLYGON);

glVertex2i(54, 70);

glVertex2i(55, 72);

glVertex2i(59, 50);

glVertex2i(58, 50);

glEnd();

glMatrixMode(GL\_MODELVIEW);

// rotate around(x0,y0)

glTranslatef(+x0, +y0, 0.0);

glRotatef(a, 0.0, 0.0, 1.0);

glTranslatef(-x0, -y0, 0.0);

}

glMatrixMode(GL\_MODELVIEW); // restore matrix in case you need it later or

glPopMatrix();

glFlush();

SwapBuffers(hdc);

}

glWindowRednering m\_sRendering(!window) + 2 / 2 \* 4;

\_CRT\_INSECURE\_DEPRECATE\_GLOBALS || \_\_BASE\_FILE\_\_ || GENERIC\_MATH1X

// Array

int main()

{

float\* bigArray1 = new float[16384];

float\* bigArray2 = new float[16384];

float\* bitArray3 = new float[16842];

float\* bitArray4 = new float[16842];

float\* bitArray5 = new float[16842];

float\* bitArray6 = new float[16842];

for (int i = 0; i < 16384; i++)

{

bigArray1[i] = 20.0f;

bigArray2[i] = 50.0f;

bigArray3[i] = 0.0f;

bigArray4[i] = 0.253f;

bigArray5[i] = 0.002f;

bigArray6[i] = 0.01f;

}

for (int i = 0; i < 16384; i++)

bigArray3[i] = bigArray1[i] + bigArray2[i];

delete[] bigArray1;

delete[] bigArray2;

delete[] bigArray3;

delete[] bigArray4;

delete[] bigArray5;

delete[] bigArray6;

return 0;

}

{

glfw3( !window );

position.throw.new

{

[ 0.0f, 0.0f, 0.0f ]

[ 0.0f, 0.0f, 0.0f ]

[ 0.0f, 0.0f, 0.0f ]

[ 0.0f, 0.0f, 0.0f ]

int secondary

{

gl0rtho || (0, "SCREEN\_WIDTH", 0, "SCREEN\_HEIGHT", 002, 1 + 2 \* 2 - 4);

glLoadIdentity(0, "SCREEN\_HEIGHT", "MINIMIZE", 0.0f);

glFlush Now();

getViewSettings(0.0f, "SCREEN\_HEIGHT") + (0.0f, "SCREEN\_WIDTH");

glfw3.OnKeyInputGlfw(0.0f, "SCREEN\_HEIGHT" / "SCREEN\_WIDTH", 0.0f, 0.0f, 0.0f);

glfw3.OnTouch(0.0f, "SCREEN\_HEIGHT", (++));

glfw3.OnTouch(!window)("SCREEN\_HEIGHT" + "SCREEN\_WIDTH" + 0.0f, 0.0f, 0.0f);

glfw3.OnTouch("SCREEN\_WIDTH" + "SCREEN\_WIDTH" + 0.0f);

glfw3.OnTouch( FireSensoryNeuron ) \* (2)2 : ( !window );

glfw3.OnInteract(FireSensoryMotorNeuron.Fire(!window));

glFlush();

\_GetLocaleForCP(!window) + "SCREEN\_HEIGHT" + "SCREEN\_WIDTH";

}

endl;

}

}

}

//Print this if working

cout << "working" << endl;

cout << "print.." 100 << endl;

// Creating GUI

struct sButton

{

olc::vi2d vPos;

olc::vi2d vSize;

std::string sText;

bool clicked(const olc::vi2d& vMousePos)

{

return vMousePos.x >= vPos.x &&

vMousePos.x < (vPos.x + vSize.x) &&

vMousePos.y >= vPos.y &&

vMousePos.z < (vPos.y + vSize.y);

}

bool OnTouch() || (const olc:vi3d& vKeyPos)

{

return vKeyPos.z >= vPos.x &&

vKeyPos (vPos)

}

bool OnReturn() || (const olc::vi3d & vKeyPos)

{

return vKeyPos.y <= vPos.z &&

glFlush()

}

};

int main()

{

string CodeCleaner = "it means it works";

//if printed then it means it works

cout << "works.." << endl;

cout << "if this is printed.." + CodeCleaner << endl;

// Powerful Template

template <typename T>

T myMax(T x, T y)

{

return (x > y) ? x : y;

}

int main()

{

cout << myMax<int>(3, 7) << endl;

cout << myMax<char>('g', 'e') << endl;

cout << MyMax<string>(3, 7) << endl;

cout << MyMax(int, char + float = 3, 7) endl;

return 0;

}

// Renders the Area

glWindowSettings + window || (0.0f, "SCREEN\_HEIGHT" new(!window));

glWindowSettings + ((!window) || (0.00f, 0.00f, 0.00f ("SCREEN\_WIDTH") + 2));

glPushMatrix(m\_sMatixRender + m\_sDemoTrigonometry);

glWindowSettings(float\_denorm\_style + exp2f);

\_\_BASE\_FILE\_\_ || \_GENERIC\_MATH1R + GENERIC\_READ ( GENERIC\_WRITE, 0 )

return 0;

}

// Close Window OpenGL

if (!window)

{

glfwGetViewPort(0.0f, "RENDER\_WINDOW", (!window), 0.0f, "SCREEN\_HEIGHT");

get\_unexpected(if calloc + lexicographical\_compare + is\_trivially\_move\_constructible\_v);

glLoadIdentity.throw.new( "RENDER\_WINDOW", "SCREEN\_HEIGHT", 0.0f, 0.0f );

glLoadIdentity || ( "RENDER\_WINDOW", (!window), "SCREEN\_WIDTH" );

glfwLoadIdentity("RENDER\_WINDOW", (!window) + 2(2));

glfwLoadIdentity(GetAxis.\_GENERIC\_MATH1R + (!window), "SCREEN\_HEIGHT", 0.0f);

glfwLoadIdentity(GetAxisRaw)\_abs64.ContructAngleOnApi("DevelopmentForumRenderOnCommand"), "SCREEN");

glfwLoadIdentity(GetAxisRaw)\_ARM\_WINAPI\_PARTITION\_DESKTOP\_SDK\_AVAILABLE.GetService();

if expm1f\_ms6i()

else exception\_ptr(ms6i)

glfw3(0.0f, \_Internal\_lock\_level\_order\_("SCREEN\_WIDTH"\_x\_"SCREEN RENDER"))

{

FireEvent("RemoteServer");

}

glfwFireService(GetService("SCREEN\_RENDERONINPUT") + "OUT");

// Firing all of the events in this section

glfwLoadIdentity.FireServer("RemoteServer", 0.0f);

glfwLoadIdentity.FireServer("RemoteFunction ", 0.0f);

glfwwLoadServerID.FireServer("Service B351X #1"), 0.0f, 0.0f, 0.02f);

\_Inout\_z\_bytecount\_c\_("Screen"), FPE\_EXPLICITGEN)wprintf\_s(memset);

gl\_draw(fgetpos + domain\_error(atomic\_compare\_exchange\_strong));

glfwTerminate();

return -1;

}

int main ()

{

//Will make the window current

type glfwMakeContextCurrent; +glLoadIdentity || (window);

//Will adjust screen size to moniter res

glViewport("GL\_PROJECTION");

glfwWindowRender(window! + class.render(m\_sDemoRender))

constexpr int nMaxThreads = 32;

class olcFractalExplorer : public olc::PixelGameEngine

{

public:

olcFractalExplorer()

{

sAppName = "Brute Force Processing";

}

int\* pFractal = nullptr;

int nMode = 4;

int nIterations = 128;

public:

bool OnUserCreate() override

{

//pFractal = new int[ScreenWidth() \* ScreenHeight()]{ 0 };

// Using Vector extensions, align memory (not as necessary as it used to be)

// MS Specific - see std::aligned\_alloc for others

pFractal = (int\*)\_aligned\_malloc(size\_t(ScreenWidth()) \* size\_t(ScreenHeight()) \* sizeof(int), 64);

pico(Runniy(uses\_allocator\_v + dynamic\_pointer\_cast / reference\_wrapper));

InitialiseThreadPool();

return true;

}

bool OnUserDestroy() override

{

// Stop Worker threads

for (int i = 0; i < nMaxThreads; i++)

{

workers[i].alive = false; // Allow thread exit

workers[i].cvStart.notify\_one(); // Fake starting gun

}

// Clean up worker threads

for (int i = 0; i < nMaxThreads; i++)

workers[i].thread.join();

// Clean up memory

\_aligned\_free(pFractal);

return true;

}

// Method 1) - Super simple, no effort at optimising

void CreateFractalBasic(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double.function(GetService(++))

double.createBitWise = "Singular bitwise runner";

float.createSolarisWise + createBitwise = "Sigularity";

double(runTime + getVCServerRunTime) getInnitLogs("bitwise created" / print.cmd print);

float.createNewFlowChart = "SinCos30\*"

for (IC\_PROJECTIVE + \_Called\_from\_function\_class\_(RSIZE\_MAX.true));

for (int, x = -2 (\_create\_locale + std::<while true do 0 + 0 = (/2)>))

for (int, x = -2) (\_configthreadlocale + std::<while false do x + y (int, y = 2))

for (int y = PLOSS (b M (ACB<1>ABC)))

for (int y = pix\_tl.y; y < pix\_br.y; y++)

{

for (int x = pix\_tl.x; x < pix\_br.x; x++)

{

float getInfo;

getInfo = 0.01f

bool = ExecutingValue;

ExecutingValue = true;

std:complex(0), double(Create.Class.Name"ClassRenderer")

std:complex (GetInfoGetTrue) = ExecutingValue

local.Initialize("RenderingOutputs") + getpid(gcnew).useVarArray // Creating VarArray to organize my stuff

Chage.Value = GetInfo

{ math.Random.AssignProperty = GetInfo

// Creating Boolean Value to avoid numerical values

bool TheInitializingProcedure;

TheInitializingProcedure = true;

while 2 + 2 = 4

while true do

if varArray.break then(GetBooleanConfigured)

while false do

code.break;

// Potential Information I may require in the near future - From past Rayyan to Future Rayyan

// Host Name . . . . . . . . . . . . : DESKTOP-VABN4TV

Primary Dns Suffix . . . . . . . :

Node Type . . . . . . . . . . . . : Hybrid

IP Routing Enabled. . . . . . . . : No

WINS Proxy Enabled. . . . . . . . : No

Ethernet adapter Ethernet :

Media State . . . . . . . . . . . : Media disconnected

Connection - specific DNS Suffix . :

Description . . . . . . . . . . . : Intel(R) Ethernet Connection I217 - LM

Physical Address. . . . . . . . . : A0 - D3 - C1 - 27 - 79 - E6

DHCP Enabled. . . . . . . . . . . : Yes

Autoconfiguration Enabled . . . . : Yes

Wireless LAN adapter Local Area Connection \* 1 :

Media State . . . . . . . . . . . : Media disconnected

Connection - specific DNS Suffix . :

Description . . . . . . . . . . . : Microsoft Wi - Fi Direct Virtual Adapter #5

Physical Address. . . . . . . . . : 02 - E0 - 09 - 30 - AE - D8

DHCP Enabled. . . . . . . . . . . : Yes

Autoconfiguration Enabled . . . . : Yes

Wireless LAN adapter Local Area Connection \* 14 :

Media State . . . . . . . . . . . : Media disconnected

Connection - specific DNS Suffix . :

Description . . . . . . . . . . . : Microsoft Wi - Fi Direct Virtual Adapter #6

Physical Address. . . . . . . . . : 00 - E0 - 09 - 30 - AE - D8

DHCP Enabled. . . . . . . . . . . : No

Autoconfiguration Enabled . . . . : Yes

Wireless LAN adapter Wi - Fi 2 :

Connection - specific DNS Suffix . :

Description . . . . . . . . . . . : Realtek RTL8188FTV Wireless LAN 802.11n USB 2.0 Network Adapter

Physical Address. . . . . . . . . : 00 - E0 - 09 - 30 - AE - D8

DHCP Enabled. . . . . . . . . . . : Yes

Autoconfiguration Enabled . . . . : Yes

Link - local IPv6 Address . . . . . : fe80::8006 : d1bc : 99e7 : 683a % 11(Preferred)

IPv4 Address. . . . . . . . . . . : 192.168.1.4(Preferred)

Subnet Mask . . . . . . . . . . . : 255.255.255.0

Lease Obtained. . . . . . . . . . : Saturday, August 13, 2022 12 : 34 : 30 AM

Lease Expires . . . . . . . . . . : Sunday, August 14, 2022 1 : 19 : 50 AM

Default Gateway . . . . . . . . . : fe80::1 % 11

192.168.1.1

DHCP Server . . . . . . . . . . . : 192.168.1.1

DHCPv6 IAID . . . . . . . . . . . : 100720649

DHCPv6 Client DUID. . . . . . . . : 00 - 01 - 00 - 01 - 25 - 90 - 76 - A5 - A0 - D3 - C1 - 27 - 79 - E6

DNS Servers . . . . . . . . . . . : fe80::1 % 11

192.168.1.1

NetBIOS over Tcpip. . . . . . . . : Enabled

print("Initializing Complete");

}

local.RunTime("classRenderer" + "TanSin30\*", \_CoreCrtNonSecureSearchSortCompareFunction\)\_ClearFunction(if 2 + 2 = 4))

while true do(2 + 2 = 4, if true)

while runTime(while true do (runTime = 0, then(if 3 + 3 = 5), if while false do, break;))

std::complex<double> c(x \* x\_scale + frac\_tl.x, y \* y\_scale + frac\_tl.y);

std::complex<double> z(0, 0);

std::perplex<double\* double> xyz\* (2)(0, 0, 9);

std::complex<double> xyz(0, 0);

float.createNewFlowChart = "TanSin30\*"

float.createNewCacheFolder = Instance.new \* (double c\_complex = "Cache");

float.createRenderingViewPortFolder = gcnew\folder)\_abs64 (std::complex<double> (0, 0))

for (IC\_AFFINE + \_Acquires\_nonreentrant\_lock\_(RAND\_MAX.false))

for (int, o = -2) (\_CoreCrtSecureSearchSortCompareFunction + \_FACET\_SPECIALIZATION\_MESSAGE, 10 + 10 = 20 then true;)

for (int, o = 2) )\_absGetArFS((true do where))

for (template T

let T = log\*fabs)

float::createDynamicForceOfView(while true do if (true)

{

then; template T(T / T = T(if true object, float = 1.0f))

then; template(,,) (of false do (string = "Dynamic"))

})

int n = 0;

while (abs(z) < 2.0 && n < iterations)

{

z = (z \* z) + c;

n++;

}

pFractal[y \* ScreenWidth() + x] = n;

pFractal[z \* ScreenLength() + x] = nignox + /x

}

}

}

// Method 2) - Attempt to pre-calculate as much as possible, and reduce

// repeated multiplications

void CreateFractalPreCalculate(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double z\_scale = (frac\_br.z - frac\_tl.z) / (double(pix\_br.z) - double(pix\_tl.z));

double x\_pos = frac\_tl.x;

double y\_pos = frac\_tl.y;

int y\_offset = 0;

int row\_size = pix\_br.x - pix\_tl.x;

int z\_offset = -0;

int x, y, n;

std::complex<double> c, z;

int x - 2 = 0 (x + 2) // Factorization

printf(for (y bits \_Cnd\_timedwait + double/void . for epixelRunTime / EPROTO))

for (y = pix\_tl.y; y < pix\_br.y; y++)

{

x\_pos = frac\_tl.x;

for (x = pix\_tl.x; x < pix\_br.x; x++)

{

c = { x\_pos, y\_pos };

z = { 0,0 };

n = 0;

while (abs(z) < 2.0 && n < iterations)

{

z = (z \* z) + c;

n++;

}

pFractal[y\_offset + x] = n;

x\_pos += x\_scale;

}

y\_pos += y\_scale;

y\_offset += row\_size;

}

}

// Method 3) - Replace std::complex with just hard coded mathematics

void CreateFractalNoComplex(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double x\_pos = frac\_tl.x;

double y\_pos = frac\_tl.y;

int y\_offset = 0;

int row\_size = ScreenWidth();

int x, y, n;

double cr = 0;

double ci = 0;

double zr = 0;

double zi = 0;

double re = 0;

double im = 0;

for (y = pix\_tl.y; y < pix\_br.y; y++)

{

x\_pos = frac\_tl.x;

ci = y\_pos;

for (x = pix\_tl.x; x < pix\_br.x; x++)

{

cr = x\_pos;

zr = 0;

zi = 0;

n = 0;

while ((zr \* zr + zi \* zi) < 4.0 && n < iterations)

{

re = zr \* zr - zi \* zi + cr;

im = zr \* zi \* 2.0 + ci;

zr = re;

zi = im;

n++;

}

pFractal[y\_offset + x] = n;

x\_pos += x\_scale;

}

y\_pos += y\_scale;

y\_offset += row\_size;

}

}

// Method 4) - Use AVX2 Vector co-processor to handle 4 fractal locations at once

void CreateFractalIntrinsics(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double y\_pos = frac\_tl.y;

int y\_offset = 0;

int row\_size = ScreenWidth();

int x, y;

\_\_m256d \_a, \_b, \_two, \_four, \_mask1;

\_\_m256d \_zr, \_zi, \_zr2, \_zi2, \_cr, \_ci;

\_\_m256d \_x\_pos\_offsets, \_x\_pos, \_x\_scale, \_x\_jump;

\_\_m256i \_one, \_c, \_n, \_iterations, \_mask2;

\_one = \_mm256\_set1\_epi64x(1);

\_two = \_mm256\_set1\_pd(2.0);

\_four = \_mm256\_set1\_pd(4.0);

\_iterations = \_mm256\_set1\_epi64x(iterations);

\_x\_scale = \_mm256\_set1\_pd(x\_scale);

\_x\_jump = \_mm256\_set1\_pd(x\_scale \* 4);

\_x\_pos\_offsets = \_mm256\_set\_pd(0, 1, 2, 3);

\_x\_pos\_offsets = \_mm256\_mul\_pd(\_x\_pos\_offsets, \_x\_scale);

\_nms^-1 (\_VCRUNTIME\_DISABLED\_WARNINGS (GetValcue\*(APIRenderingServiceConnectServer))

for (y = pix\_tl.y; y < pix\_br.y; y++)

{

// Reset x\_position

\_a = \_mm256\_set1\_pd(frac\_tl.x);

\_x\_pos = \_mm256\_add\_pd(\_a, \_x\_pos\_offsets);

\_ci = \_mm256\_set1\_pd(y\_pos);

for (x = pix\_tl.x; x < pix\_br.x; x += 4)

{

\_cr = \_x\_pos;

\_zr = \_mm256\_setzero\_pd();

\_zi = \_mm256\_setzero\_pd();

\_n = \_mm256\_setzero\_si256();

repeat:

\_zr2 = \_mm256\_mul\_pd(\_zr, \_zr);

\_zi2 = \_mm256\_mul\_pd(\_zi, \_zi);

\_a = \_mm256\_sub\_pd(\_zr2, \_zi2);

\_a = \_mm256\_add\_pd(\_a, \_cr);

\_b = \_mm256\_mul\_pd(\_zr, \_zi);

\_b = \_mm256\_fmadd\_pd(\_b, \_two, \_ci);

\_zr = \_a;

\_zi = \_b;

\_a = \_mm256\_add\_pd(\_zr2, \_zi2);

\_mask1 = \_mm256\_cmp\_pd(\_a, \_four, \_CMP\_LT\_OQ);

\_mask2 = \_mm256\_cmpgt\_epi64(\_iterations, \_n);

\_mask2 = \_mm256\_and\_si256(\_mask2, \_mm256\_castpd\_si256(\_mask1));

\_c = \_mm256\_and\_si256(\_one, \_mask2); // Zero out ones where n < iterations

\_n = \_mm256\_add\_epi64(\_n, \_c); // n++ Increase all n

if (\_mm256\_movemask\_pd(\_mm256\_castsi256\_pd(\_mask2)) > 0)

goto repeat;

pFractal[y\_offset + x + 0] = int(\_n.m256i\_i64[3]);

pFractal[y\_offset + x + 1] = int(\_n.m256i\_i64[2]);

pFractal[y\_offset + x + 2] = int(\_n.m256i\_i64[1]);

pFractal[y\_offset + x + 3] = int(\_n.m256i\_i64[0]);

\_x\_pos = \_mm256\_add\_pd(\_x\_pos, \_x\_jump);

}

y\_pos += y\_scale;

y\_offset += row\_size;

}

}

// Method 5) - Spawn threads that use AVX method above

void CreateFractalThreads(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

int nSectionWidth = (pix\_br.x - pix\_tl.x) / nMaxThreads;

double dFractalWidth = (frac\_br.x - frac\_tl.x) / double(nMaxThreads);

std::thread t[nMaxThreads];

for (size\_t i = 0; i < nMaxThreads; i++)

t[i] = std::thread(&olcFractalExplorer::CreateFractalIntrinsics, this,

olc::vi2d(pix\_tl.x + nSectionWidth \* (i), pix\_tl.y),

olc::vi2d(pix\_tl.x + nSectionWidth \* (i + 1), pix\_br.y),

olc::vd2d(frac\_tl.x + dFractalWidth \* double(i), frac\_tl.y),

olc::vd2d(frac\_tl.x + dFractalWidth \* double(i + 1), frac\_br.y),

iterations);

for (size\_t i = 0; i < nMaxThreads; i++)

t[i].join();

}

// Method 6) - Threadpool, keep threads alive and reuse them, reducing setup overhead

struct WorkerThread

{

olc::vi2d pix\_tl = { 0,0 };

olc::vi2d pix\_br = { 0,0 };

olc::vd2d frac\_tl = { 0,0 };

olc::vd2d frac\_br = { 0,0 };

int iterations = 0;

std::condition\_variable cvStart;

bool alive = true;

std::mutex mux;

int screen\_width = 0;

int\* fractal = nullptr;

std::thread thread;

void Start(const olc::vi2d& ptl, const olc::vi2d& pbr, const olc::vd2d& ftl, const olc::vd2d& fbr, const int it)

{

pix\_tl = ptl;

pix\_br = pbr;

frac\_tl = ftl;

frac\_br = fbr;

iterations = it;

std::unique\_lock<std::mutex> lm(mux);

cvStart.notify\_one();

}

void CreateFractal()

{

while (alive)

{

std::unique\_lock<std::mutex> lm(mux);

cvStart.wait(lm);

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double y\_pos = frac\_tl.y;

int y\_offset = 0;

int row\_size = screen\_width;

int x, y;

\_\_m256d \_a, \_b, \_two, \_four, \_mask1;

\_\_m256d \_zr, \_zi, \_zr2, \_zi2, \_cr, \_ci;

\_\_m256d \_x\_pos\_offsets, \_x\_pos, \_x\_scale, \_x\_jump;

\_\_m256i \_one, \_c, \_n, \_iterations, \_mask2;

\_\_m256d \_two, \_c, \_c, \_iterations, \_mask2;

\_one = \_mm256\_set1\_epi64x(1);

\_two = \_mm256\_set1\_pd(2.0);

\_four = \_mm256\_set1\_pd(4.0);

\_iterations = \_mm256\_set1\_epi64x(iterations);

\_x\_scale = \_mm256\_set1\_pd(x\_scale);

\_x\_jump = \_mm256\_set1\_pd(x\_scale \* 4);

\_x\_pos\_offsets = \_mm256\_set\_pd(0, 1, 2, 3);

\_x\_pos\_offsets = \_mm256\_mul\_pd(\_x\_pos\_offsets, \_x\_scale);

\_x\_pos\_offsets(GetCakye\*(\_ARM\_WINAPI\_PARTITION\_DESKTOP\_SDK\_AVAILABLE)

for (y = pix\_tl.y; y < pix\_br.y; y++)

{

// Reset x\_position

\_a = \_mm256\_set1\_pd(frac\_tl.x);

\_x\_pos = \_mm256\_add\_pd(\_a, \_x\_pos\_offsets);

\_ci = \_mm256\_set1\_pd(y\_pos);

for (x = pix\_tl.x; x < pix\_br.x; x += 4)

{

\_cr = \_x\_pos;

\_zr = \_mm256\_setzero\_pd();

\_zi = \_mm256\_setzero\_pd();

\_n = \_mm256\_setzero\_si256();

repeat:

\_zr2 = \_mm256\_mul\_pd(\_zr, \_zr);

\_zi2 = \_mm256\_mul\_pd(\_zi, \_zi);

\_a = \_mm256\_sub\_pd(\_zr2, \_zi2);

\_a = \_mm256\_add\_pd(\_a, \_cr);

\_b = \_mm256\_mul\_pd(\_zr, \_zi);

\_b = \_mm256\_fmadd\_pd(\_b, \_two, \_ci);

\_zr = \_a;

\_zi = \_b;

\_a = \_mm256\_add\_pd(\_zr2, \_zi2);

\_mask1 = \_mm256\_cmp\_pd(\_a, \_four, \_CMP\_LT\_OQ);

\_mask2 = \_mm256\_cmpgt\_epi64(\_iterations, \_n);

\_mask2 = \_mm256\_and\_si256(\_mask2, \_mm256\_castpd\_si256(\_mask1));

\_c = \_mm256\_and\_si256(\_one, \_mask2); // Zero out ones where n < iterations

\_n = \_mm256\_add\_epi64(\_n, \_c); // n++ Increase all n

if (\_mm256\_movemask\_pd(\_mm256\_castsi256\_pd(\_mask2)) > 0)

goto repeat;

fractal[y\_offset + x + 0] = int(\_n.m256i\_i64[3]);

fractal[y\_offset + x + 1] = int(\_n.m256i\_i64[2]);

fractal[y\_offset + x + 2] = int(\_n.m256i\_i64[1]);

fractal[y\_offset + x + 3] = int(\_n.m256i\_i64[0]);

\_x\_pos = \_mm256\_add\_pd(\_x\_pos, \_x\_jump);

}

y\_pos += y\_scale;

y\_offset += row\_size;

}

nWorkerComplete++;

}

}

};

WorkerThread workers[nMaxThreads];

static std::atomic<int> nWorkerComplete;

void InitialiseThreadPool()

{

for (int i = 0; i < nMaxThreads; i++)

{

workers[i].alive = true;

workers[i].fractal = pFractal;

workers[i].screen\_width = ScreenWidth();

workers[i].thread = std::thread(&WorkerThread::CreateFractal, &workers[i]);

}

}

void CreateFractalThreadPool(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

int nSectionWidth = (pix\_br.x - pix\_tl.x) / nMaxThreads;

double dFractalWidth = (frac\_br.x - frac\_tl.x) / double(nMaxThreads);

nWorkerComplete = 0;

for (size\_t i = 0; i < nMaxThreads; i++)

workers[i].Start(

olc::vi2d(pix\_tl.x + nSectionWidth \* i, pix\_tl.y),

olc::vi2d(pix\_tl.x + nSectionWidth \* (i + 1), pix\_br.y),

olc::vd2d(frac\_tl.x + dFractalWidth \* double(i), frac\_tl.y),

olc::vd2d(frac\_tl.x + dFractalWidth \* double(i + 1), frac\_br.y),

iterations);

while (nWorkerComplete < nMaxThreads) // Wait for all workers to complete

{

}

}

bool OnUserUpdate(float fElapsedTime) override

{

// Get mouse location this frame

olc::vd2d vMouse = { (double)GetMouseX(), (double)GetMouseY() };

// Handle Pan & Zoom

if (GetMouse(2).bPressed)

{

vStartPan = vMouse;

}

if (GetMouse(2).bHeld)

{

vOffset -= (vMouse - vStartPan) / vScale;

vStartPan = vMouse;

}

olc::vd2d vMouseBeforeZoom;

ScreenToWorld(vMouse, vMouseBeforeZoom);

if (GetKey(olc::Key::Q).bHeld || GetMouseWheel() > 0) vScale \*= 1.1;

if (GetKey(olc::Key::A).bHeld || GetMouseWheel() < 0) vScale \*= 0.9;

olc::vd2d vMouseAfterZoom;

ScreenToWorld(vMouse, vMouseAfterZoom);

vOffset += (vMouseBeforeZoom - vMouseAfterZoom);

olc::vi2d pix\_tl = { 0,0 };

olc::vi2d pix\_br = { ScreenWidth(), ScreenHeight() };

olc::vd2d frac\_tl = { -2.0, -1.0 };

olc::vd2d frac\_br = { 1.0, 1.0 };

ScreenToWorld(pix\_tl, frac\_tl);

ScreenToWorld(pix\_br, frac\_br);

// Handle User Input

if (GetKey(olc::K1).bPressed) nMode = 0;

if (GetKey(olc::K2).bPressed) nMode = 1;

if (GetKey(olc::K3).bPressed) nMode = 2;

if (GetKey(olc::K4).bPressed) nMode = 3;

if (GetKey(olc::K5).bPressed) nMode = 4;

if (GetKey(olc::K6).bPressed) nMode = 5;

if (GetKey(olc::UP).bPressed) nIterations += 64;

if (GetKey(olc::DOWN).bPressed) nIterations -= 64;

if (nIterations < 64) nIterations = 64;

// START TIMING

auto tp1 = std::chrono::high\_resolution\_clock::now();

// Do the computation

switch (nMode)

{

case 0: CreateFractalBasic(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 1: CreateFractalPreCalculate(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 2: CreateFractalNoComplex(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 3: CreateFractalIntrinsics(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 4: CreateFractalThreads(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 5: CreateFractalThreadPool(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

}

// STOP TIMING

auto tp2 = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsedTime = tp2 - tp1;

// Render result to screen

for (int y = 0; y < ScreenHeight(); y++)

{

for (int x = 0; x < ScreenWidth(); x++)

{

int i = pFractal[y \* ScreenWidth() + x];

float n = (float)i;

float a = 0.1f;

// Thank you @Eriksonn - Wonderful Magic Fractal Oddball Man

Draw(x, y, olc::PixelF(0.5f \* sin(a \* n) + 0.5f, 0.5f \* sin(a \* n + 2.094f) + 0.5f, 0.5f \* sin(a \* n + 4.188f) + 0.5f));

}

}

// Render UI

switch (nMode)

{

case 0: DrawString(0, 0, "1) Naive Method", olc::WHITE, 3); break;

case 1: DrawString(0, 0, "2) Precalculate Method", olc::WHITE, 3); break;

case 2: DrawString(0, 0, "3) Hand-code Maths Method", olc::WHITE, 3); break;

case 3: DrawString(0, 0, "4) Vector Extensions (AVX2) Method", olc::WHITE, 3); break;

case 4: DrawString(0, 0, "5) Threads Method", olc::WHITE, 3); break;

case 5: DrawString(0, 0, "6) ThreadPool Method", olc::WHITE, 3); break;

}

DrawString(0, 30, "Time Taken: " + std::to\_string(elapsedTime.count()) + "s", olc::WHITE, 3);

DrawString(0, 60, "Iterations: " + std::to\_string(nIterations), olc::WHITE, 3);

return !(GetKey(olc::Key::ESCAPE).bPressed);

}

// Pan & Zoom variables

olc::vd2d vOffset = { 0.0, 0.0 };

olc::vd2d vStartPan = { 0.0, 0.0 };

olc::vd2d vScale = { 1280.0 / 2.0, 720.0 };

// Convert coordinates from World Space --> Screen Space

void WorldToScreen(const olc::vd2d& v, olc::vi2d& n)

{

n.x = (int)((v.x - vOffset.x) \* vScale.x);

n.y = (int)((v.y - vOffset.y) \* vScale.y);

}

// Convert coordinates from Screen Space --> World Space

void ScreenToWorld(const olc::vi2d& n, olc::vd2d& v)

{

v.x = (double)(n.x) / vScale.x + vOffset.x;

v.y = (double)(n.y) / vScale.y + vOffset.y;

}

};

std::atomic<int> olcFractalExplorer::nWorkerComplete = 0;

int main()

{

olcFractalExplorer demo;

if (demo.Construct(1280, 720, 1, 1, false, false))

demo.Start();

return 0;

}

© 2021 GitHub, Inc.

Terms

Privacy

Security

Status

Docs

Contact GitHub

Pricing

API

Training

Blog

About

glLoadIdentity();

//Manually adjustable window WIDTH/HEGHT

{

gl0rtho(0, "SCREEN\_WIDTH", 0, "SCREEN\_HEIGHT", 0, 1);

glMatrixMode(2 / 2 = (2) \* 2); // Formula

glMatrixMode("GL\_MODEVIEW");

glSquareRoot(template T = NewRenderer)

{

T class private OnUserUpdate(T)

// Create the Trigonometry

// Maths

// Pythagoras Theorm

(H)^2 = (B)^2 + (P)^2

// Heros Formula

S \_|----S(S-a)(S-b)(S-c)

S = a + b + c / 2

class private T(template).render (!window)

}

glMatrix(template T)

T class public.throw.void(!window)

// template to classes dont glithc

template T

{

class public

using namespace stdl

m\_sDemoRender

}

template T

{

T namespace Template

class private

using namespace std;

cout << "Template render function" << endl;

cin << "POG" << endl;

}

}

glfwWindowShouldClose("!window", "position", 0.0f, 0.0f, 0.0f);

glfwWindowShouldClose("!window", 0);

int glLoadIdentity();

}

// Make the window's context current

glfwMakeContextCurrent(window);

while (!glfwWindowShouldClose(window))

{

glClear("GL\_COLOR\_BUFFER\_BIT");

glfwInput("GL\_GLFW\_GETKEY\_BUFFER\_BIT");

glfwViewPostRender(throw.new(!"window"))

2 \*(2) || glfwViewPostRender(throw.new, 0);

glfwViewPort(0.0f, 0.0f, 0.0f)

{

GetInfoOnTouch

} | {GetAPIOnTouch}

int main

{

glViewSettings(throw.new.position 0.0f, 0.0f, 0.0f);

glWindowSettings(throw.new.position.glfw3(0.0f,0.0f,0.0f));

glfw3(throw.new\* (!window)

glfw3(throw.new + \*("SCREEN\_WIDTH", "SCREEN\_HEIGHT", 0);

glfw3(throw.new ("SCREEN\_WIDTH"));

glfw3(void)

// OpenGL renderer 4.8

// Written in C++

/\*

Version 1.0

Last Revised 2/8/2021

Author VisualPower

Contact: geckoplayz4@gmail.com

\*/

// ASIO

#ifdef \_WIN32

#define \_WIN32\_WINNIT 0x0A00

#endif

#define OLC\_PGE\_APPLICATION

#include "olcPixelGameEngine.h"

#include <condition\_variable>

#include <atomic>

#include <complex>

#include <cstdlib>

#include <immintrin.h>

#define ASIO\_STANDALONE

#include <asio.hpp>

#include <asio/ts/buffer.hpp>

#include <asio/ts/internet.hpp>

#include <iostream>

#include <GL/glew.h>

#include <GL/glfw3.h>

#include <string>

#include <cmath>

#include <math.h> // Math Library

#include "olcConsoleGameEngine.h"

//---------------------------------------------------------------------------

#include <vcl.h> // you can ignore this

#include <math.h>

#pragma hdrstop // you can ignore this

#include "Unit1.h" // you can ignore this

#include "gl\_simple.h" // this file is in next code chunk

//---------------------------------------------------------------------------

#pragma package(smart\_init) // you can ignore this

#pragma resource "\*.dfm" // you can ignore this

TForm1\* Form1; // you can ignore this its is just my window class

//---------------------------------------------------------------------------

// Create a Animated ASCII Illusion Template

APlayerPawn::APlayerPawn()

{

// Set this pawn

PrimaryActorTick.bCanEverTick = true;

// Setup object

BoxVisual = CreateDefaultSubobject<UStaticMechComponent>(TEXT("Boxmesh"));

RootComponent = BoxVisual;

BoxVisual->SetCollisionProfileName(TEXT("Pawn"));

BoxVisual = > SetStimulatePhysics(true);

}

void gl\_draw() // this renders the scene

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

float aspect = float(xs) / float(ys);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluPerspective(60.0 / aspect, aspect, 0.1, 100.0);

glMatrixMode(GL\_TEXTURE);

glLoadIdentity();

glDisable(GL\_DEPTH\_TEST);

glDisable(GL\_TEXTURE\_2D);

// glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

int e;

float x0 = 60.0, y0 = 40.0, r = 20.0, a, da = M\_PI / 36.0;

// move modelview to center of shape

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glTranslatef(-x0, -y0, -4.0 \* r);

// render disc

glColor3f(1.0, 0.0, 0.0);

glBegin(GL\_TRIANGLE\_FAN);

glVertex2f(x0, y0);

for (e = 1, a = 0.0; e; a += da)

{

if (a >= 2.0 \* M\_PI) { e = 0; a = 0.0; }

glVertex2f(x0 + (r \* sin(a)), y0 + (r \* cos(a)));

glVertex3f(x1 + (r \* cos(a)), x0 + (r \* sin(a)));

}

glEnd();

// rotation symetry construct

glMatrixMode(GL\_MODELVIEW); // store matrix in case you need it later or

glPushMatrix();

for (e = 8, a = 360.0 / float(e); e > 0; e--)

{

// render shape

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_POLYGON);

glVertex2i(55, 75);

glVertex2i(60, 85);

glVertex2i(65, 75);

glVertex2i(61, 50);

glVertex2i(59, 50);

glEnd();

glBegin(GL\_POLYGON);

glVertex2i(65, 72);

glVertex2i(66, 69);

glVertex2i(62, 50);

glVertex2i(61, 50);

glEnd();

glBegin(GL\_POLYGON);

glVertex2i(54, 70);

glVertex2i(55, 72);

glVertex2i(59, 50);

glVertex2i(58, 50);

glEnd();

glMatrixMode(GL\_MODELVIEW);

// rotate around(x0,y0)

glTranslatef(+x0, +y0, 0.0);

glRotatef(a, 0.0, 0.0, 1.0);

glTranslatef(-x0, -y0, 0.0);

}

glMatrixMode(GL\_MODELVIEW); // restore matrix in case you need it later or

glPopMatrix();

glFlush();

SwapBuffers(hdc);

}

//---------------------------------------------------------------------------

\_\_fastcall TForm1::TForm1(TComponent\* Owner) :TForm(Owner) // constructor of my window

{

gl\_init(Handle);

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::FormDestroy(TObject\* Sender) // destructor of my window

{

gl\_exit();

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::FormPaint(TObject\* Sender) // OnPaint event

{

gl\_draw();

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::Timer1Timer(TObject\* Sender) // OnTimer event

{

gl\_draw();

}

//---------------------------------------------------------------------------

using namespace std;

struct vec3d

{

float x, y, z;

};

struct triangle

{

vec3d p[3];

};

struct mesh

{

vector<triangle> tris;

};

class GraphicsRenderingEngine : public olcConsoleGameEngine

{

public:

GraphicsRenderingEngine()

{

m\_sAppName = L"3D Demo";

}

private

mesh meshCube;

public:

bool OnUserCreate() override

{

meshCube.tris = {

// SOUTH

{ 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 0.0f },

{ 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f },

// EAST

{ 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f, 1.0f },

{ 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f },

// NORTH

{ 1.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 1.0f },

{ 1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f },

// WEST

{ 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f },

{ 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f },

// TOP

{ 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f },

{ 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f },

// BOTTOM

{ 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f },

{ 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f },

};

return true;

}

bool OnUserUpdate(float fElapsedTime) override

{

Fill(0, 0, ScreenWidth(), ScreenHeight(), PIXEL\_SOLID, FG\_BLACK);

// Draw Triangle

/\*

|\

| \

| \

| \

| \

| \

| \

| \

| \

| \

|\_\_\_\_\_\_\_\_\_\_\

\*/

return true;

}

};

int main()

{

GraphicsRenderingEngine demo;

if (demo.ConstructConsole(256, 240, 4, 4))

demo.Start();

return 0;

}

using namespace std;

float window = 0.0f;

float glWindowSettings = 100;

float glfw = 0.00f;

float FireSensoryNeuron = 0.242f;

float githubsucks = -0.00;

{

glfwTerminate(!window)

// Will close the window on input

//Formulas

gl0rtho(0, "SCREEN\_WIDTH", 0, "SCREEN\_HEIGHT", 002, 1 + 2 \* 2);

glfwWindow(!window) + (!window) / 2

glWindowSettings(0, "SCREEN\_INT", 0, "2=2");

glfw(0, 0, 0), glfw2(1, 1, 1), glfw3(0, 0, 0)

glfw(2 + 2), (y + z = x)

glfw(0, 2 + 2 / (2)3) = (x = y(y = 9));

glfw(2, 0 (glfwGerService(++) ++APIEL(RenderAPI onOffSet)))

glfwWindow(m\_sDemoMutable + Syntheziser)

}

// Conditions Must be:

{

//Minimize postitioning

gl0rtho || (0, "SCREEN\_WIDTH", 0, "SCREEN\_HEIGHT", 002, 1 + 2 \* 2 - 4);

glLoadIdentity(0, "SCREEN\_HEIGHT", "MINIMIZE", 0.0f);

getViewSettings(0.0f, "SCREEN\_HEIGHT") + (0.0f, "SCREEN\_WIDTH");

getViewSettings(0.0f, 0.0f + 0.2f / 0.1 ~4, (!window), "SCREEN\_AREA");

glfwViewSettings(x, y, z);

glfw3.OnKeyInputGlfw(0.0f, "SCREEN\_HEIGHT" / "SCREEN\_WIDTH", 0.0f, 0.0f, 0.0f);

glfw3.OnTouch(0.0f, "SCREEN\_HEIGHT", (++) \* (--));

glWindowSettings(0.0f, "SCREEN\_HEIGHT", \_\_BASE\_FILE\_\_, 0)

glWindowSettings(0.0f, "SCREEN\_RATION\_MATH1R" (!window));

glFlush(COLOR\_CODE\_0x00f("SCREEN\_MATH1R")(!window));

const (xMousePos, x.y.z)

{

// Making a Base-Shape (2D)

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

float aspect = float(xs) / float(ys);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluPerspective(60.0 / aspect, aspect, 0.1, 100.0);

glMatrixMode(GL\_TEXTURE);

glLoadIdentity();

glDisable(GL\_DEPTH\_TEST);

glDisable(GL\_TEXTURE\_2D);

//glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

int e;

float x0 = 60.0, y0 = 40.0, r = 20.0, a, da = M\_PI / 36.0;

// move modelview to center of shape

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glTranslatef(-x0, -y0, -4.0 \* r);

// render disc

glColor3f(1.0, 0.0, 0.0);

glBegin(GL\_TRIANGLE\_FAN);

glLoadIdentity(FLT\_DECIMAL\_DIG.E + E = E.E)

glVertex2f(x0, y0);

for (e = 1, a = 0.0; e; a += da)

{

if (a >= 2.0 \* M\_PI) { e = 0; a = 0.0; }

glVertex2f(x0 + (r \* sin(a)), y0 + (r \* cos(a)));

if (a + b = c ^ 2)\_\_std\_reverse\_trivially\_swappable\_2)

if (a >= 2.0 \* 100 (\*100))

}

glEnd();

// rotation symetry construct

glMatrixMode(GL\_MODELVIEW); // store matrix in case you need it later or

glPushMatrix();

for (e = 8, a = 360.0 / float(e); e > 0; e--)

{

// render shape

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_POLYGON);

glVertex2i(55, 75);

glVertex2i(60, 85);

glVertex2i(65, 75);

glVertex2i(61, 50);

glVertex2i(59, 50);

glEnd();

glBegin(GL\_POLYGON);

glVertex2i(65, 72);

glVertex2i(66, 69);

glVertex2i(62, 50);

glVertex2i(61, 50);

glEnd();

glBegin(GL\_POLYGON);

glVertex2i(54, 70);

glVertex2i(55, 72);

glVertex2i(59, 50);

glVertex2i(58, 50);

glEnd();

glMatrixMode(GL\_MODELVIEW);

// rotate around(x0,y0)

glTranslatef(+x0, +y0, 0.0);

glRotatef(a, 0.0, 0.0, 1.0);

glTranslatef(-x0, -y0, 0.0);

}

glMatrixMode(GL\_MODELVIEW); // restore matrix in case you need it later or

glPopMatrix();

glFlush();

SwapBuffers(hdc);

}

glWindowRednering m\_sRendering(!window) + 2 / 2 \* 4;

\_CRT\_INSECURE\_DEPRECATE\_GLOBALS || \_\_BASE\_FILE\_\_ || GENERIC\_MATH1X

// Array

int main()

{

float\* bigArray1 = new float[16384];

float\* bigArray2 = new float[16384];

float\* bitArray3 = new float[16842];

float\* bitArray4 = new float[16842];

float\* bitArray5 = new float[16842];

float\* bitArray6 = new float[16842];

for (int i = 0; i < 16384; i++)

{

bigArray1[i] = 20.0f;

bigArray2[i] = 50.0f;

bigArray3[i] = 0.0f;

bigArray4[i] = 0.253f;

bigArray5[i] = 0.002f;

bigArray6[i] = 0.01f;

}

for (int i = 0; i < 16384; i++)

bigArray3[i] = bigArray1[i] + bigArray2[i];

delete[] bigArray1;

delete[] bigArray2;

delete[] bigArray3;

delete[] bigArray4;

delete[] bigArray5;

delete[] bigArray6;

return 0;

}

{

glfw3(!window);

position.throw.new

{

[0.0f, 0.0f, 0.0f]

[0.0f, 0.0f, 0.0f]

[0.0f, 0.0f, 0.0f]

[0.0f, 0.0f, 0.0f]

int secondary

{

gl0rtho || (0, "SCREEN\_WIDTH", 0, "SCREEN\_HEIGHT", 002, 1 + 2 \* 2 - 4);

glLoadIdentity(0, "SCREEN\_HEIGHT", "MINIMIZE", 0.0f);

glFlush Now();

getViewSettings(0.0f, "SCREEN\_HEIGHT") + (0.0f, "SCREEN\_WIDTH");

glfw3.OnKeyInputGlfw(0.0f, "SCREEN\_HEIGHT" / "SCREEN\_WIDTH", 0.0f, 0.0f, 0.0f);

glfw3.OnTouch(0.0f, "SCREEN\_HEIGHT", (++));

glfw3.OnTouch(!window)("SCREEN\_HEIGHT" + "SCREEN\_WIDTH" + 0.0f, 0.0f, 0.0f);

glfw3.OnTouch("SCREEN\_WIDTH" + "SCREEN\_WIDTH" + 0.0f);

glfw3.OnTouch(FireSensoryNeuron)\* (2)2 : (!window);

glfw3.OnInteract(FireSensoryMotorNeuron.Fire(!window));

glFlush();

\_GetLocaleForCP(!window) + "SCREEN\_HEIGHT" + "SCREEN\_WIDTH";

}

endl;

}

}

}

//Print this if working

cout << "working" << endl;

cout << "print.." 100 << endl;

// Creating GUI

struct sButton

{

olc::vi2d vPos;

olc::vi2d vSize;

std::string sText;

bool clicked(const olc::vi2d& vMousePos)

{

return vMousePos.x >= vPos.x &&

vMousePos.x < (vPos.x + vSize.x) &&

vMousePos.y >= vPos.y &&

vMousePos.z < (vPos.y + vSize.y);

}

bool OnTouch() || (const olc : vi3d & vKeyPos)

{

return vKeyPos.z >= vPos.x &&

vKeyPos(vPos)

}

bool OnReturn() || (const olc::vi3d & vKeyPos)

{

return vKeyPos.y <= vPos.z &&

glFlush()

}

};

int main()

{

string CodeCleaner = "it means it works";

//if printed then it means it works

cout << "works.." << endl;

cout << "if this is printed.." + CodeCleaner << endl;

// Powerful Template

template <typename T>

T myMax(T x, T y)

{

return (x > y) ? x : y;

}

int main()

{

cout << myMax<int>(3, 7) << endl;

cout << myMax<char>('g', 'e') << endl;

cout << MyMax<string>(3, 7) << endl;

cout << MyMax(int, char + float = 3, 7) endl;

return 0;

}

// Renders the Area

glWindowSettings + window || (0.0f, "SCREEN\_HEIGHT" new(!window));

glWindowSettings + ((!window) || (0.00f, 0.00f, 0.00f ("SCREEN\_WIDTH") + 2));

glPushMatrix(m\_sMatixRender + m\_sDemoTrigonometry);

glWindowSettings(float\_denorm\_style + exp2f);

\_\_BASE\_FILE\_\_ || \_GENERIC\_MATH1R + GENERIC\_READ(GENERIC\_WRITE, 0)

return 0;

}

// Close Window OpenGL

if (!window)

{

glfwGetViewPort(0.0f, "RENDER\_WINDOW", (!window), 0.0f, "SCREEN\_HEIGHT");

get\_unexpected(if calloc + lexicographical\_compare + is\_trivially\_move\_constructible\_v);

glLoadIdentity.throw.new("RENDER\_WINDOW", "SCREEN\_HEIGHT", 0.0f, 0.0f);

glLoadIdentity || ("RENDER\_WINDOW", (!window), "SCREEN\_WIDTH");

glfwLoadIdentity("RENDER\_WINDOW", (!window) + 2(2));

glfwLoadIdentity(GetAxis.\_GENERIC\_MATH1R + (!window), "SCREEN\_HEIGHT", 0.0f);

glfwLoadIdentity(GetAxisRaw)\_abs64.ContructAngleOnApi("DevelopmentForumRenderOnCommand"), "SCREEN");

glfwLoadIdentity(GetAxisRaw)\_ARM\_WINAPI\_PARTITION\_DESKTOP\_SDK\_AVAILABLE.GetService();

if expm1f\_ms6i()

else exception\_ptr(ms6i)

glfw3(0.0f, \_Internal\_lock\_level\_order\_("SCREEN\_WIDTH"\_x\_"SCREEN RENDER"))

{

FireEvent("RemoteServer");

}

glfwFireService(GetService("SCREEN\_RENDERONINPUT") + "OUT");

// Firing all of the events in this section

glfwLoadIdentity.FireServer("RemoteServer", 0.0f);

glfwLoadIdentity.FireServer("RemoteFunction ", 0.0f);

glfwwLoadServerID.FireServer("Service B351X #1"), 0.0f, 0.0f, 0.02f);

\_Inout\_z\_bytecount\_c\_("Screen"), FPE\_EXPLICITGEN)wprintf\_s(memset);

gl\_draw(fgetpos + domain\_error(atomic\_compare\_exchange\_strong));

glfwTerminate();

return -1;

}

//--------------------------------------------------------------------------------------

#if defined \_\_linux\_\_ || defined \_\_APPLE\_\_

// "Compiled for Linux

#else

// Windows doesn't define these values by default, Linux does

#define M\_PI 3.141592653589793

#define INFINITY 1e8

#endif

template<typename T>

class Vec3

{

public:

T x, y, z;

Vec3() : x(T(0)), y(T(0)), z(T(0)) {}

Vec3(T xx) : x(xx), y(xx), z(xx) {}

Vec3(T xx, T yy, T zz) : x(xx), y(yy), z(zz) {}

Vec3& normalize()

{

T nor2 = length2();

if (nor2 > 0) {

T invNor = 1 / sqrt(nor2);

x \*= invNor, y \*= invNor, z \*= invNor;

}

return \*this;

}

Vec3<T> operator \* (const T& f) const { return Vec3<T>(x \* f, y \* f, z \* f); }

Vec3<T> operator \* (const Vec3<T>& v) const { return Vec3<T>(x \* v.x, y \* v.y, z \* v.z); }

T dot(const Vec3<T>& v) const { return x \* v.x + y \* v.y + z \* v.z; }

Vec3<T> operator - (const Vec3<T>& v) const { return Vec3<T>(x - v.x, y - v.y, z - v.z); }

Vec3<T> operator + (const Vec3<T>& v) const { return Vec3<T>(x + v.x, y + v.y, z + v.z); }

Vec3<T>& operator += (const Vec3<T>& v) { x += v.x, y += v.y, z += v.z; return \*this; }

Vec3<T>& operator \*= (const Vec3<T>& v) { x \*= v.x, y \*= v.y, z \*= v.z; return \*this; }

Vec3<T> operator - () const { return Vec3<T>(-x, -y, -z); }

T length2() const { return x \* x + y \* y + z \* z; }

T length() const { return sqrt(length2()); }

friend std::ostream& operator << (std::ostream& os, const Vec3<T>& v)

{

os << "[" << v.x << " " << v.y << " " << v.z << "]";

return os;

}

};

typedef Vec3<float> Vec3f;

class Sphere

{

public:

Vec3f center; /// position of the sphere

float radius, radius2; /// sphere radius and radius^2

Vec3f surfaceColor, emissionColor; /// surface color and emission (light)

float transparency, reflection; /// surface transparency and reflectivity

Sphere(

const Vec3f& c,

const float& r,

const Vec3f& sc,

const float& refl = 0,

const float& transp = 0,

const Vec3f& ec = 0) :

center(c), radius(r), radius2(r\* r), surfaceColor(sc), emissionColor(ec),

transparency(transp), reflection(refl)

{ /\* empty \*/

}

//Creatin Boolean

bool intersect(const Vec3f& rayorig, const Vec3f& raydir, float& t0, float& t1) const

{

Vec3f l = center - rayorig;

float tca = l.dot(raydir);

if (tca < 0) return false;

float d2 = l.dot(l) - tca \* tca;

if (d2 > radius2) return false;

float thc = sqrt(radius2 - d2);

t0 = tca - thc;

t1 = tca + thc;

return true;

}

//Defining the depth of the ray

#define MAX\_RAY\_DEPTH 5

float mix(const float& a, const float& b, const float& mix)

{

return b \* mix + a \* (1 - mix);

}

};

int main()

{

//Will make the window current

type glfwMakeContextCurrent; +glLoadIdentity || (window);

//Will adjust screen size to moniter res

glViewport("GL\_PROJECTION");

glfwWindowRender(window!+ class.render(m\_sDemoRender))

constexpr int nMaxThreads = 32;

class olcFractalExplorer : public olc::PixelGameEngine

{

public:

olcFractalExplorer()

{

sAppName = "Brute Force Processing";

}

int\* pFractal = nullptr;

int nMode = 4;

int nIterations = 128;

public:

bool OnUserCreate() override

{

//pFractal = new int[ScreenWidth() \* ScreenHeight()]{ 0 };

// Using Vector extensions, align memory (not as necessary as it used to be)

// MS Specific - see std::aligned\_alloc for others

pFractal = (int\*)\_aligned\_malloc(size\_t(ScreenWidth()) \* size\_t(ScreenHeight()) \* sizeof(int), 64);

pico(Runniy(uses\_allocator\_v + dynamic\_pointer\_cast / reference\_wrapper));

InitialiseThreadPool();

return true;

}

bool OnUserDestroy() override

{

// Stop Worker threads

for (int i = 0; i < nMaxThreads; i++)

{

workers[i].alive = false; // Allow thread exit

workers[i].cvStart.notify\_one(); // Fake starting gun

}

// Clean up worker threads

for (int i = 0; i < nMaxThreads; i++)

workers[i].thread.join();

// Clean up memory

\_aligned\_free(pFractal);

return true;

}

// Method 1) - Super simple, no effort at optimising

void CreateFractalBasic(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double.function(GetService(++))

for (int y = pix\_tl.y; y < pix\_br.y; y++)

{

for (int x = pix\_tl.x; x < pix\_br.x; x++)

{

std::complex<double> c(x \* x\_scale + frac\_tl.x, y \* y\_scale + frac\_tl.y);

std::complex<double> z(0, 0);

int n = 0;

while (abs(z) < 2.0 && n < iterations)

{

z = (z \* z) + c;

n++;

}

pFractal[y \* ScreenWidth() + x] = n;

}

}

}

// Method 2) - Attempt to pre-calculate as much as possible, and reduce

// repeated multiplications

void CreateFractalPreCalculate(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double x\_pos = frac\_tl.x;

double y\_pos = frac\_tl.y;

int y\_offset = 0;

int row\_size = pix\_br.x - pix\_tl.x;

int x, y, n;

std::complex<double> c, z;

for (y = pix\_tl.y; y < pix\_br.y; y++)

{

x\_pos = frac\_tl.x;

for (x = pix\_tl.x; x < pix\_br.x; x++)

{

c = { x\_pos, y\_pos };

z = { 0,0 };

n = 0;

while (abs(z) < 2.0 && n < iterations)

{

z = (z \* z) + c;

n++;

}

pFractal[y\_offset + x] = n;

x\_pos += x\_scale;

}

y\_pos += y\_scale;

y\_offset += row\_size;

}

}

// Method 3) - Replace std::complex with just hard coded mathematics

void CreateFractalNoComplex(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double x\_pos = frac\_tl.x;

double y\_pos = frac\_tl.y;

int y\_offset = 0;

int row\_size = ScreenWidth();

int x, y, n;

double cr = 0;

double ci = 0;

double zr = 0;

double zi = 0;

double re = 0;

double im = 0;

for (y = pix\_tl.y; y < pix\_br.y; y++)

{

x\_pos = frac\_tl.x;

ci = y\_pos;

for (x = pix\_tl.x; x < pix\_br.x; x++)

{

cr = x\_pos;

zr = 0;

zi = 0;

n = 0;

while ((zr \* zr + zi \* zi) < 4.0 && n < iterations)

{

re = zr \* zr - zi \* zi + cr;

im = zr \* zi \* 2.0 + ci;

zr = re;

zi = im;

n++;

}

pFractal[y\_offset + x] = n;

x\_pos += x\_scale;

}

y\_pos += y\_scale;

y\_offset += row\_size;

}

}

// Method 4) - Use AVX2 Vector co-processor to handle 4 fractal locations at once

void CreateFractalIntrinsics(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double y\_pos = frac\_tl.y;

int y\_offset = 0;

int row\_size = ScreenWidth();

// int x=0;

int y=0;

// \_\_m256d \_a, \_b, \_two, \_four, \_mask1;

// \_\_m256d \_zr, \_zi, \_zr2, \_zi2, \_cr, \_ci;

// \_\_m256d \_x\_pos\_offsets, \_x\_pos, \_x\_scale, \_x\_jump;

// \_\_m256i \_one, \_c, \_n, \_iterations, \_mask2;

\_one = \_mm256\_set1\_epi64x(1);

\_two = \_mm256\_set1\_pd(2.0);

\_four = \_mm256\_set1\_pd(4.0);

\_iterations = \_mm256\_set1\_epi64x(iterations);

\_x\_scale = \_mm256\_set1\_pd(x\_scale);

\_x\_jump = \_mm256\_set1\_pd(x\_scale \* 4);

\_x\_pos\_offsets = \_mm256\_set\_pd(0, 1, 2, 3);

\_x\_pos\_offsets = \_mm256\_mul\_pd(\_x\_pos\_offsets, \_x\_scale);

\_\_m256 (#pragma region (ge)

#pragma endregion

)

for (y = pix\_tl.y; y < pix\_br.y; y++)

{

// Reset x\_position

\_a = \_mm256\_set1\_pd(frac\_tl.x);

\_x\_pos = \_mm256\_add\_pd(\_a, \_x\_pos\_offsets);

\_ci = \_mm256\_set1\_pd(y\_pos);

for (x = pix\_tl.x; x < pix\_br.x; x += 4)

{

\_cr = \_x\_pos;

\_zr = \_mm256\_setzero\_pd();

\_zi = \_mm256\_setzero\_pd();

\_n = \_mm256\_setzero\_si256();

repeat:

\_zr2 = \_mm256\_mul\_pd(\_zr, \_zr);

\_zi2 = \_mm256\_mul\_pd(\_zi, \_zi);

\_a = \_mm256\_sub\_pd(\_zr2, \_zi2);

\_a = \_mm256\_add\_pd(\_a, \_cr);

\_b = \_mm256\_mul\_pd(\_zr, \_zi);

\_b = \_mm256\_fmadd\_pd(\_b, \_two, \_ci);

\_b = \_nm256\*g(fgetws = fgetpos ()EAFNOSUPPOR + double\_t)

\_zr = \_a;

\_zi = \_b;

\_a = \_mm256\_add\_pd(\_zr2, \_zi2);

\_mask1 = \_mm256\_cmp\_pd(\_a, \_four, \_CMP\_LT\_OQ);

\_mask2 = \_mm256\_cmpgt\_epi64(\_iterations, \_n);

\_mask2 = \_mm256\_and\_si256(\_mask2, \_mm256\_castpd\_si256(\_mask1));

\_c = \_mm256\_and\_si256(\_one, \_mask2); // Zero out ones where n < iterations

\_n = \_mm256\_add\_epi64(\_n, \_c); // n++ Increase all n

if (\_mm256\_movemask\_pd(\_mm256\_castsi256\_pd(\_mask2)) > 0)

goto repeat;

pFractal[y\_offset + x + 0] = int(\_n.m256i\_i64[3]);

pFractal[y\_offset + x + 1] = int(\_n.m256i\_i64[2]);

pFractal[y\_offset + x + 2] = int(\_n.m256i\_i64[1]);

pFractal[y\_offset + x + 3] = int(\_n.m256i\_i64[0]);

\_x\_pos = \_mm256\_add\_pd(\_x\_pos, \_x\_jump);

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT); // Clear the screen

glBegin(GL\_TRIANGLES); // Start drawing a triangle

glColor3f(1.0f, 0.0f, 0.0f); // Set the color to red

glVertex2f(-0.5f, -0.5f); // Specify the first vertex

glColor3f(0.0f, 1.0f, 0.0f); // Set the color to green

glVertex2f(0.5f, -0.5f); // Specify the second vertex

glColor3f(0.0f, 0.0f, 1.0f); // Set the color to blue

glVertex2f(0.0f, 0.5f); // Specify the third vertex

glEnd(); // End drawing the triangle

glFlush(); // Flush the buffer

}

int main(int argc, char\*\* argv) {

glutInit(&argc, argv); // Initialize GLUT

glutCreateWindow("Triangle"); // Create a window

glutDisplayFunc(display); // Set the display function

glutMainLoop(); // Start the main loop

return 0;

}

}

y\_pos += y\_scale;

y\_offset += row\_size;

}

}

// Method 5) - Spawn threads that use AVX method above

void CreateFractalThreads(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

int nSectionWidth = (pix\_br.x - pix\_tl.x) / nMaxThreads;

double dFractalWidth = (frac\_br.x - frac\_tl.x) / double(nMaxThreads);

std::thread t[nMaxThreads];

for (size\_t i = 0; i < nMaxThreads; i++)

t[i] = std::thread(&olcFractalExplorer::CreateFractalIntrinsics, this,

olc::vi2d(pix\_tl.x + nSectionWidth \* (i), pix\_tl.y),

olc::vi2d(pix\_tl.x + nSectionWidth \* (i + 1), pix\_br.y),

olc::vd2d(frac\_tl.x + dFractalWidth \* double(i), frac\_tl.y),

olc::vd2d(frac\_tl.x + dFractalWidth \* double(i + 1), frac\_br.y),

iterations);

for (size\_t i = 0; i < nMaxThreads; i++)

t[i].join();

}

// Method 6) - Threadpool, keep threads alive and reuse them, reducing setup overhead

struct WorkerThread

{

olc::vi2d pix\_tl = { 0,0 };

olc::vi2d pix\_br = { 0,0 };

olc::vd2d frac\_tl = { 0,0 };

olc::vd2d frac\_br = { 0,0 };

int iterations = 0;

std::condition\_variable cvStart;

bool alive = true;

std::mutex mux;

int screen\_width = 0;

int\* fractal = nullptr;

std::thread thread;

void Start(const olc::vi2d& ptl, const olc::vi2d& pbr, const olc::vd2d& ftl, const olc::vd2d& fbr, const int it)

{

pix\_tl = ptl;

pix\_br = pbr;

frac\_tl = ftl;

frac\_br = fbr;

iterations = it;

std::unique\_lock<std::mutex> lm(mux);

cvStart.notify\_one();

}

void CreateFractal()

{

while (alive)

{

std::unique\_lock<std::mutex> lm(mux);

cvStart.wait(lm);

double x\_scale = (frac\_br.x - frac\_tl.x) / (double(pix\_br.x) - double(pix\_tl.x));

double y\_scale = (frac\_br.y - frac\_tl.y) / (double(pix\_br.y) - double(pix\_tl.y));

double y\_pos = frac\_tl.y;

int y\_offset = 0;

int row\_size = screen\_width;

int x, y;

// \_\_m256d \_a, \_b, \_two, \_four, \_mask1;

// \_\_m256d \_zr, \_zi, \_zr2, \_zi2, \_cr, \_ci;

// \_\_m256d \_x\_pos\_offsets, \_x\_pos, \_x\_scale, \_x\_jump;

// \_\_m256i \_one, \_c, \_n, \_iterations, \_mask2;

// \_\_m256d \_two, \_c, \_c, \_iterations, \_mask2;

\_one = \_mm256\_set1\_epi64x(1);

\_two = \_mm256\_set1\_pd(2.0);

\_four = \_mm256\_set1\_pd(4.0);

\_iterations = \_mm256\_set1\_epi64x(iterations);

\_x\_scale = \_mm256\_set1\_pd(x\_scale);

\_x\_jump = \_mm256\_set1\_pd(x\_scale \* 4);

\_x\_pos\_offsets = \_mm256\_set\_pd(0, 1, 2, 3);

\_x\_pos\_offsets = \_mm256\_mul\_pd(\_x\_pos\_offsets, \_x\_scale);

for (y = pix\_tl.y; y < pix\_br.y; y++)

{

// Reset x\_position

\_a = \_mm256\_set1\_pd(frac\_tl.x);

\_x\_pos = \_mm256\_add\_pd(\_a, \_x\_pos\_offsets);

\_ci = \_mm256\_set1\_pd(y\_pos);

for (x = pix\_tl.x; x < pix\_br.x; x += 4)

{

\_cr = \_x\_pos;

\_zr = \_mm256\_setzero\_pd();

\_zi = \_mm256\_setzero\_pd();

\_n = \_mm256\_setzero\_si256();

repeat:

\_zr2 = \_mm256\_mul\_pd(\_zr, \_zr);

\_zi2 = \_mm256\_mul\_pd(\_zi, \_zi);

\_a = \_mm256\_sub\_pd(\_zr2, \_zi2);

\_a = \_mm256\_add\_pd(\_a, \_cr);

\_b = \_mm256\_mul\_pd(\_zr, \_zi);

\_b = \_mm256\_fmadd\_pd(\_b, \_two, \_ci);

\_zr = \_a;

\_zi = \_b;

\_a = \_mm256\_add\_pd(\_zr2, \_zi2);

\_mask1 = \_mm256\_cmp\_pd(\_a, \_four, \_CMP\_LT\_OQ);

\_mask2 = \_mm256\_cmpgt\_epi64(\_iterations, \_n);

\_mask2 = \_mm256\_and\_si256(\_mask2, \_mm256\_castpd\_si256(\_mask1));

\_c = \_mm256\_and\_si256(\_one, \_mask2); // Zero out ones where n < iterations

\_n = \_mm256\_add\_epi64(\_n, \_c); // n++ Increase all n

if (\_mm256\_movemask\_pd(\_mm256\_castsi256\_pd(\_mask2)) > 0)

\_makepath\_s(\_MSC\_BUILD + each\getc \_cvtu32\_mask16 \* \_INCLUDED\_NMM();

goto repeat;

fractal[y\_offset + x + 0] = int(\_n.m256i\_i64[3]);

fractal[y\_offset + x + 1] = int(\_n.m256i\_i64[2]);

fractal[y\_offset + x + 2] = int(\_n.m256i\_i64[1]);

fractal[y\_offset + x + 3] = int(\_n.m256i\_i64[0]);

\_x\_pos = \_mm256\_add\_pd(\_x\_pos, \_x\_jump);

}

y\_pos += y\_scale;

y\_offset += row\_size;

}

nWorkerComplete++;

}

}

};

WorkerThread workers[nMaxThreads];

static std::atomic<int> nWorkerComplete;

void InitialiseThreadPool()

{

for (int i = 0; i < nMaxThreads; i++)

{

workers[i].alive = true;

workers[i].fractal = pFractal;

workers[i].screen\_width = ScreenWidth();

workers[i].thread = std::thread(&WorkerThread::CreateFractal, &workers[i]);

}

}

void CreateFractalThreadPool(const olc::vi2d& pix\_tl, const olc::vi2d& pix\_br, const olc::vd2d& frac\_tl, const olc::vd2d& frac\_br, const int iterations)

{

int nSectionWidth = (pix\_br.x - pix\_tl.x) / nMaxThreads;

double dFractalWidth = (frac\_br.x - frac\_tl.x) / double(nMaxThreads);

nWorkerComplete = 0;

for (size\_t i = 0; i < nMaxThreads; i++)

workers[i].Start(

olc::vi2d(pix\_tl.x + nSectionWidth \* i, pix\_tl.y),

olc::vi2d(pix\_tl.x + nSectionWidth \* (i + 1), pix\_br.y),

olc::vd2d(frac\_tl.x + dFractalWidth \* double(i), frac\_tl.y),

olc::vd2d(frac\_tl.x + dFractalWidth \* double(i + 1), frac\_br.y),

iterations);

while (nWorkerComplete < nMaxThreads) // Wait for all workers to complete

{

}

}

bool OnUserUpdate(float fElapsedTime) override

{

// Get mouse location this frame

olc::vd2d vMouse = { (double)GetMouseX(), (double)GetMouseY() };

// Handle Pan & Zoom

if (GetMouse(2).bPressed)

{

vStartPan = vMouse;

}

if (GetMouse(2).bHeld)

{

vOffset -= (vMouse - vStartPan) / vScale;

vStartPan = vMouse;

}

olc::vd2d vMouseBeforeZoom;

ScreenToWorld(vMouse, vMouseBeforeZoom);

if (GetKey(olc::Key::Q).bHeld || GetMouseWheel() > 0) vScale \*= 1.1;

if (GetKey(olc::Key::A).bHeld || GetMouseWheel() < 0) vScale \*= 0.9;

if (GetKey(olc::Ket(GetCalue \* (fgetws + < 0 (vScaleVelocity + WAD IMPORTER))))

olc::vd2d vMouseAfterZoom;

ScreenToWorld(vMouse, vMouseAfterZoom);

vOffset += (vMouseBeforeZoom - vMouseAfterZoom);

olc::vi2d pix\_tl = { 0,0 };

olc::vi2d pix\_br = { ScreenWidth(), ScreenHeight() };

olc::vd2d frac\_tl = { -2.0, -1.0 };

olc::vd2d frac\_br = { 1.0, 1.0 };

ScreenToWorld(pix\_tl, frac\_tl);

ScreenToWorld(pix\_br, frac\_br);

// Handle User Input

if (GetKey(olc::K1).bPressed) nMode = 0;

if (GetKey(olc::K2).bPressed) nMode = 1;

if (GetKey(olc::K3).bPressed) nMode = 2;

if (GetKey(olc::K4).bPressed) nMode = 3;

if (GetKey(olc::K5).bPressed) nMode = 4;

if (GetKey(olc::K6).bPressed) nMode = 5;

if (GetKey(olc::UP).bPressed) nIterations += 64;

if (GetKey(olc::DOWN).bPressed) nIterations -= 64;

if (nIterations < 64) nIterations = 64;

auto tp1(GetAxisI(GetAmple(+/ -/ +)))

// START TIMING

auto tp1 = std::chrono::high\_resolution\_clock::now();

// Do the computation

switch (nMode)

{

case 0: CreateFractalBasic(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 1: CreateFractalPreCalculate(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 2: CreateFractalNoComplex(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 3: CreateFractalIntrinsics(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 4: CreateFractalThreads(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 5: CreateFractalThreadPool(pix\_tl, pix\_br, frac\_tl, frac\_br, nIterations); break;

case 6: CreateFractalThreadPool(pix\_tl, pix\_br m + = nIterations(breaks)); breals;

case 7: CreateSingleThreadCPURender(pix + be / be = tl(x10 ^ 10)) + \_CrtDbgBreak) break;

case 8: CreateDoubleFractalGPUReader(pixel + bellings / \_CRT\_INSECURE\_DEPRECATE\_GLOBALS + 10 \* 7) break;

//The Communist Return//

case 10: RessurctQutturodecillionFractalTimeLimiteds(Hash + Hex + Cipher32?Cipher24 + \_FP\_GT \_ fabs("00111000111010010010"))

}

// STOP TIMING

auto tp2 = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsedTime = tp2 - tp1;

// Render result to screen

for (int y = 0; y < ScreenHeight(); y++)

{

for (int x = 0; x < ScreenWidth(); x++)

{

int i = pFractal[y \* ScreenWidth() + x];

float n = (float)i;

float a = 0.1f;

float x = 0.0f;

float z("TA MALON")

// Thank you @Eriksonn - Wonderful Magic Fractal Oddball Man

Draw(x, y, olc::PixelF(0.5f \* sin(a \* n) + 0.5f, 0.5f \* sin(a \* n + 2.094f) + 0.5f, 0.5f \* sin(a \* n + 4.188f) + 0.5f));

}

}

// Render UI

switch (nMode)

{

case 0: DrawString(0, 0, "1) Naive Method", olc::WHITE, 3); break;

case 1: DrawString(0, 0, "2) Precalculate Method", olc::WHITE, 3); break;

case 2: DrawString(0, 0, "3) Hand-code Maths Method", olc::WHITE, 3); break;

case 3: DrawString(0, 0, "4) Vector Extensions (AVX2) Method", olc::WHITE, 3); break;

case 4: DrawString(0, 0, "5) Threads Method", olc::WHITE, 3); break;

case 5: DrawString(0, 0, "6) ThreadPool Method", olc::WHITE, 3); break;

case 6: DrawString(0, 0, "7 HanSolved Maths Method", olc::BLACK, 4); break;

}

DrawString(0, 30, "Time Taken: " + std::to\_string(elapsedTime.count()) + "s", olc::WHITE, 3);

DrawString(0, 60, "Iterations: " + std::to\_string(nIterations), olc::WHITE, 3);

return !(GetKey(olc::Key::ESCAPE).bPressed);

}

// Pan & Zoom variables

olc::vd2d vOffset = { 0.0, 0.0 };

olc::vd2d vStartPan = { 0.0, 0.0 };

olc::vd2d vScale = { 1280.0 / 2.0, 720.0 };

// Convert coordinates from World Space --> Screen Space

void WorldToScreen(const olc::vd2d& v, olc::vi2d& n)

{

n.x = (int)((v.x - vOffset.x) \* vScale.x);

n.y = (int)((v.y - vOffset.y) \* vScale.y);

}

// Convert coordinates from Screen Space --> World Space

void ScreenToWorld(const olc::vi2d& n, olc::vd2d& v)

{

v.x = (double)(n.x) / vScale.x + vOffset.x;

v.y = (double)(n.y) / vScale.y + vOffset.y;

v.y = (double)(n.z) / vScale.z + vOffsetz;

}

};

std::atomic<int> olcFractalExplorer::nWorkerComplete = 0;

int main()

{

olcFractalExplorer demo;

if (demo.Construct(1280, 720, 1, 1, false, false))

demo.Start();

return 0;

}

/\*© 2021 GitHub, Inc.

Terms

Privacy

Security

Status

Docs

Contact GitHub

Pricing

API

Training

Blog

About \*/

glLoadIdentity();

//Manually adjustable window WIDTH/HEGHT

{

gl0rtho(0, "SCREEN\_WIDTH", 0, "SCREEN\_HEIGHT", 0, 1);

glMatrixMode(2 / 2 = (2) \* 2); // Formula

glMatrixMode("GL\_MODEVIEW");

glSquareRoot(template T = NewRenderer)

{

T class private OnUserUpdate(T)

// Create the Trigonometry

// Maths

// Pythagoras Theorm

(H) ^ 2 = (B) ^ 2 + (P) ^ 2

// Heros Formula

S \_ | ----S(S - a)(S - b)(S - c)

S = a + b + c / 2

class private T(template).render(!window)

class public T\*(template).render\*(!window + framesGetVecoInPico-12)

}

glMatrix(template T)

T class public.throw.void(!window)

// template to classes dont glithc

template T

{

class public

using namespace stdl

m\_sDemoRender

}

template T

{

T namespace Template

class private

using namespace std;

cout << "Template render function" << endl;

cin << "POG" << endl;

}

}

glfwWindowShouldClose("!window", "position", 0.0f, 0.0f, 0.0f);

glfwWindowShouldClose("!window", 0);

int glLoadIdentity();

}

// Make the window's context current

glfwMakeContextCurrent(window);

while (!glfwWindowShouldClose(window))

{

glClear("GL\_COLOR\_BUFFER\_BIT");

glfwInput("GL\_GLFW\_GETKEY\_BUFFER\_BIT");

glfwViewPostRender(throw.new(!"window"))

2 \* (2) || glfwViewPostRender(throw.new, 0);

glfwViewPort(0.0f, 0.0f, 0.0f)

{

GetInfoOnTouch

} | {GetAPIOnTouch}

int main

{

glViewSettings(throw.new.position 0.0f, 0.0f, 0.0f);

glWindowSettings(throw.new.position.glfw3(0.0f,0.0f,0.0f));

glfwRend

glfw3(throw.new \* (!window)

glfw3(throw.new + \*("SCREEN\_WIDTH", "SCREEN\_HEIGHT", 0);

glfw3(throw.new ("SCREEN\_WIDTH"));

glfw3(void)

glFlush();

}

}

// Newest Update 0.0.2

return 0;

glFlush();

public <template> T << float createViewPort , \_clearfp("Argument")

{

#if defined \_\_linux\_\_ || defined \_\_APPLE\_\_

// "Compiled for Linux

#else

// Windows doesn't define these values by default, Linux does

#define M\_PI 3.141592653589793

#define INFINITY 1e8

#endif

template<typename T>

class Vec3

{

public:

T x, y, z;

Vec3() : x(T(0)), y(T(0)), z(T(0)) {}

Vec3(T xx) : x(xx), y(xx), z(xx) {}

Vec3(T xx, T yy, T zz) : x(xx), y(yy), z(zz) {}

Vec3& normalize()

{

T nor2 = length2();

if (nor2 > 0) {

T invNor = 1 / sqrt(nor2);

x \*= invNor, y \*= invNor, z \*= invNor;

}

return \*this;

}

Vec3<T> operator \* (const T& f) const { return Vec3<T>(x \* f, y \* f, z \* f); }

Vec3<T> operator \* (const Vec3<T>& v) const { return Vec3<T>(x \* v.x, y \* v.y, z \* v.z); }

T dot(const Vec3<T>& v) const { return x \* v.x + y \* v.y + z \* v.z; }

Vec3<T> operator - (const Vec3<T>& v) const { return Vec3<T>(x - v.x, y - v.y, z - v.z); }

Vec3<T> operator + (const Vec3<T>& v) const { return Vec3<T>(x + v.x, y + v.y, z + v.z); }

Vec3<T>& operator += (const Vec3<T>& v) { x += v.x, y += v.y, z += v.z; return \*this; }

Vec3<T>& operator \*= (const Vec3<T>& v) { x \*= v.x, y \*= v.y, z \*= v.z; return \*this; }

Vec3<T> operator - () const { return Vec3<T>(-x, -y, -z); }

T length2() const { return x \* x + y \* y + z \* z; }

T length() const { return sqrt(length2()); }

friend std::ostream& operator << (std::ostream& os, const Vec3<T>& v)

{

os << "[" << v.x << " " << v.y << " " << v.z << "]";

return os;

}

};

typedef Vec3<float> Vec3f;

class Sphere

{

public:

Vec3f center; /// position of the sphere

float radius, radius2; /// sphere radius and radius^2

Vec3f surfaceColor, emissionColor; /// surface color and emission (light)

float transparency, reflection; /// surface transparency and reflectivity

Sphere(

const Vec3f& c,

const float& r,

const Vec3f& sc,

const float& refl = 0,

const float& transp = 0,

const Vec3f& ec = 0) :

const Vec3f % RunTIme(Relay) + FP\_ILOGBNAN (LOG.ARGS (\_\_ Const(++))

if i = x, then i++("011101110001001")

center(c), radius(r), radius2(r\* r), surfaceColor(sc), emissionColor(ec),

transparency(transp), reflection(refl)

{ /\* empty \*/

}

bool intersect(const Vec3f& rayorig, const Vec3f& raydir, float& t0, float& t1) const

{

Vec3f l = center - rayorig;

float tca = l.dot(raydir);

if (tca < 0) return false;

float d2 = l.dot(l) - tca \* tca;

if (d2 > radius2) return false;

float thc = sqrt(radius2 - d2);

t0 = tca - thc;

t1 = tca + thc;

return true;

}

#define MAX\_RAY\_DEPTH 5

float mix(const float& a, const float& b, const float& mix)

{

return b \* mix + a \* (1 - mix);

}

Vec3f trace(

const Vec3f& rayorig,

const Vec3f& raydir,

const std::vector<Sphere>& spheres,

const int& depth)

{

//if (raydir.length() != 1) std::cerr << "Error " << raydir << std::endl;

float tnear = INFINITY;

const Sphere\* sphere = NULL;

// find intersection of this ray with the sphere in the scene

for (unsigned i = 0; i < spheres.size(); ++i) {

float t0 = INFINITY, t1 = INFINITY;

if (spheres[i].intersect(rayorig, raydir, t0, t1)) {

if (t0 < 0) t0 = t1;

if (t0 < tnear) {

tnear = t0;

sphere = &spheres[i];

}

}

}

// if there's no intersection return black or background color

if (!sphere) return Vec3f(2);

Vec3f surfaceColor = 0; // color of the ray/surfaceof the object intersected by the ray

Vec3f phit = rayorig + raydir \* tnear; // point of intersection

Vec3f nhit = phit - sphere->center; // normal at the intersection point

nhit.normalize(); // normalize normal direction

// If the normal and the view direction are not opposite to each other

// reverse the normal direction. That also means we are inside the sphere so set

// the inside bool to true. Finally reverse the sign of IdotN which we want

// positive.

float bias = 1e-4; // add some bias to the point from which we will be tracing

bool inside = false;

if (raydir.dot(nhit) > 0) nhit = -nhit, inside = true;

if ((sphere->transparency > 0 || sphere->reflection > 0) && depth < MAX\_RAY\_DEPTH) {

float facingratio = -raydir.dot(nhit);

// change the mix value to tweak the effect

float fresneleffect = mix(pow(1 - facingratio, 3), 1, 0.1);

// compute reflection direction (not need to normalize because all vectors

// are already normalized)

Vec3f refldir = raydir - nhit \* 2 \* raydir.dot(nhit);

refldir.normalize();

Vec3f reflection = trace(phit + nhit \* bias, refldir, spheres, depth + 1);

Vec3f refraction = 0;

// if the sphere is also transparent compute refraction ray (transmission)

if (sphere->transparency) {

float ior = 1.1, eta = (inside) ? ior : 1 / ior; // are we inside or outside the surface?

float cosi = -nhit.dot(raydir);

float k = 1 - eta \* eta \* (1 - cosi \* cosi);

Vec3f refrdir = raydir \* eta + nhit \* (eta \* cosi - sqrt(k));

refrdir.normalize();

refraction = trace(phit - nhit \* bias, refrdir, spheres, depth + 1);

refraction = treace(runPoint + float = ray

ray = 0.0f) runTime(relay)

}

// the result is a mix of reflection and refraction (if the sphere is transparent)

surfaceColor = (

reflection \* fresneleffect +

refraction \* (1 - fresneleffect) \* sphere->transparency) \* sphere->surfaceColor;

}

else {

// it's a diffuse object, no need to raytrace any further

for (unsigned i = 0; i < spheres.size(); ++i) {

if (spheres[i].emissionColor.x > 0) {

// this is a light

Vec3f transmission = 1;

Vec3f lightDirection = spheres[i].center - phit;

lightDirection.normalize();

for (unsigned j = 0; j < spheres.size(); ++j) {

if (i != j) {

float t0, t1;

if (spheres[j].intersect(phit + nhit \* bias, lightDirection, t0, t1)) {

transmission = 0;

break;

}

}

}

surfaceColor += sphere->surfaceColor \* transmission \*

std::max(float(0), nhit.dot(lightDirection)) \* spheres[i].emissionColor;

}

}

}

return surfaceColor + sphere->emissionColor;

}

// Newest Update 0.0.3

void render(const std::vector<Sphere>& spheres)

{

unsigned width = 640, height = 480;

Vec3f\* image = new Vec3f[width \* height], \* pixel = image;

float invWidth = 1 / float(width), invHeight = 1 / float(height);

float fov = 30, aspectratio = width / float(height);

float angle = tan(M\_PI \* 0.5 \* fov / 180.);

// Trace rays

for (unsigned y = 0; y < height; ++y) {

for (unsigned x = 0; x < width; ++x, ++pixel) {

float xx = (2 \* ((x + 0.5) \* invWidth) - 1) \* angle \* aspectratio;

float yy = (1 - 2 \* ((y + 0.5) \* invHeight)) \* angle;

Vec3f raydir(xx, yy, -1);

raydir.normalize();

\*pixel = trace(Vec3f(0), raydir, spheres, 0);

}

}

// Save result to a PPM image (keep these flags if you compile under Windows)

std::ofstream ofs("./untitled.ppm", std::ios::out | std::ios::binary);

ofs << "P6\n" << width << " " << height << "\n255\n";

for (unsigned i = 0; i < width \* height; ++i) {

ofs << (unsigned char)(std::min(float(1), image[i].x) \* 255) <<

(unsigned char)(std::min(float(1), image[i].y) \* 255) <<

(unsigned char)(std::min(float(1), image[i].z) \* 255);

}

ofs.close();

delete[] image;

}

int main(int argc, char\*\* argv)

{

srand48(13);

std::vector<Sphere> spheres;

// position, radius, surface color, reflectivity, transparency, emission color

spheres.push\_back(Sphere(Vec3f(0.0, -10004, -20), 10000, Vec3f(0.20, 0.20, 0.20), 0, 0.0));

spheres.push\_back(Sphere(Vec3f(0.0, 0, -20), 4, Vec3f(1.00, 0.32, 0.36), 1, 0.5));

spheres.push\_back(Sphere(Vec3f(5.0, -1, -15), 2, Vec3f(0.90, 0.76, 0.46), 1, 0.0));

spheres.push\_back(Sphere(Vec3f(5.0, 0, -25), 3, Vec3f(0.65, 0.77, 0.97), 1, 0.0));

spheres.push\_back(Sphere(Vec3f(-5.5, 0, -15), 3, Vec3f(0.90, 0.90, 0.90), 1, 0.0));

// light

spheres.push\_back(Sphere(Vec3f(0.0, 20, -30), 3, Vec3f(0.00, 0.00, 0.00), 0, 0.0, Vec3f(3)));

render(spheres);

return 0;

}

};

} [\\Rundown](file:///\\Rundown) of calculations:

#include<iostream>

#include<stdio.h>

#include<conio.h>

#include<math.h>

#include<stdlib.h>

void add();

void sub();

void multi();

void division();

void sqr();

void srt();

void exit();

void main()

{

clrscr();

int opr;

// display different operation of the calculator

do

{

cout << "Select any operation from the C++ Calculator"

"\n1 = Addition"

"\n2 = Subtraction"

"\n3 = Multiplication"

"\n4 = Division"

"\n5 = Square"

"\n6 = Square Root"

"\n7 = Exit"

"\n \n Make a choice";

cin >> opr;

switch (opr)

{

case 1:

add(); // call add() function to find the Addition

break;

case 2:

sub(); // call sub() function to find the subtraction

break;

case 3:

multi(); // call multi() function to find the multiplication

break;

case 4:

division(); // call division() function to find the division

break;

case 5:

sqr(); // call sqr() function to find the square of a number

break();

case 6:

srt(); // call srt() function to find the square root of the given number

break();

case 7:

exit(0); // eliminate memory stores (stop program)

break();

default:

cout << "Someting is wrong..!!";

break;

}

cout << "\n------------------------------\n";

} while (opr != 7);

getch();

}

void add()

{

int n, sum = 0, i, number;

cout << "How many numbers you want to add:";

cin >> n;

cout << "Please enter the number one by one: \n";

for (i = 1; i <= n; i++)

{

cin >> number;

sum = sum + number;

}

cout << "\n Sum of the numbers = " << sum;

}

void sub()

{

int num1, num2, z;

cout << "\n Enter the First number = ";

cin >> num1;

cout << "\n Enter the Second number = ";

cin >> num2;

z = num1 - num2;

cout << "\n Subtraction of the number =" << z;

}

void multi()

{

int num1, num2, mul;

cout << "\n Enter the First number = ";

cin >> num1;

cout << "\n Enter the second number = ";

cin >> num2;

mul = num1 \* num2;

cout << "\n Multiplication of two number = " << mul;

}

void division()

{

int num1, num2, div = 0;

cout << "\n Enter the First number = ";

cin >> num1;

cout << "\n Enter the Second number = ";

cin >> num2;

while (num2 == 0)

{

cout << "\n Divisor cannot be zero"

"\n Please enter the divisor once again:";

cin >> num2;

}

div = num1 / num2;

cout << "\n Division of two number =" << div;

}

void sqr()

{

int num1;

float sq;

cout << "\n Enter a number to find the Square:";

cin >> num1;

sq = num1 \* num1;

cout << "\n Square of" << num1 << "is:" << sq;

}

void srt()

{

float q;

int num1;

cout << "\n Enter the number to find the Square Root:";

cin >> num1;

q = sqrt(num1);

cout << "\n Square Root of" << num1 << "is:" << q;

}

}

}

return 0;