# Pointers, vectors, structs Tut12

**Tasks set in tutorials should be finished in your own time!**

## Intro

Look at the presentation and/or watch the videos on pointers, vectors and structures. We’re pulling a lot of things we’ve looked at previously together to make something closer to a game.

### Get the starter code

Get the code from tutorial 8:

[**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t08\_spritemovesol.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t08_spritemovesol.git) as our starter, a previous tutorial with a spaceship you can move around the screen. Check it compiles and runs, familiarise yourself with the code, if there’s anything you don’t understand ASK! 😊

1. Clone it as “t11\_ship”.
2. Put it on GiT.

### Create a Ship structure

Take the data for a ship and the behaviour of flying it around the screen and put it into a structure.

struct Ship

{

Sprite spr;

void Init(RenderWindow& window, Texture& tex);

void Update(const Vector2u& screenSize, float elapsedSec);

void Render(RenderWindow& window);

};

main()

{

Texture shipTex;

LoadTexture(“data/ship.png”, shipTex);

Ship ship;

ship.Init(window, shipTex);

while(window.isOpen())

{

ship.Update(window.getSize(), elapsed);

ship.Render(window);  
}

}

We’re moving into object oriented territory again here. Notice how you can “read the code” as collections of objects (structs) passing messages to each other. If the functions and variable names are good enough then it should make sense without having to read all the implementation details.

### Create an Asteroid structure

Similar to the ship, get an asteroid to render in the middle of the screen.

struct Rock {

//similar to a ship

//…..

void Update(const Vector2u& screenSize) {

spr.setPosition(screenSize.x/2.f, screenSize.y/2.f);

}

};

Rock rock;

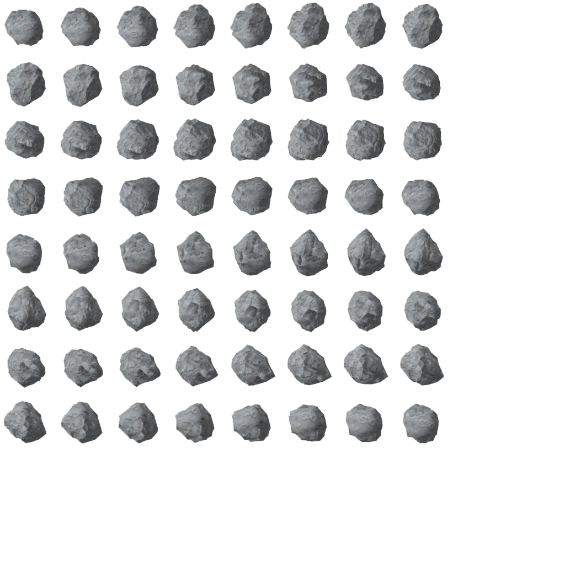
rock.Init(window, rockTex);

while(window.isOpen())

{

rock.Update(window.getSize());

rock.Render(window);  
}



Bear in mind the asteroid textures is an atlas containing two asteroid sprite animations. Going left to right, the first 32 images are one animation and the remaining 32 are another. For now, we just want the top left image.

sf::IntRect texR(0, 0, 96, 96);

spr.setTextureRect(texR);

spr.setOrigin(texR.width / 2.f, texR.height / 2.f);

If you did want to animate the asteroid, you need to switch from one image to the next, for all 32 frames of animation, at a fixed rate. Do it at about 25fps and it will look smooth, perhaps a job for later?

### Create an Object structure

There’s a lot of overlap between a ship and an asteroid: the way they look, the way they move(ish). Create a single structure called ‘Object’ and use it to represent asteroids or ships, use an enum class to indicate which job a particular Object instance has. Stop us duplicating code and allows ships and asteroids to be held in the same container (array).

struct Object

{

enum class ObjT { Ship, Rock };

ObjT type;

Sprite spr;

void Update(const Vector2u& screenSz);

void Render(RenderWindow& window);

void InitShip(RenderWindow& window, Texture& tex);

void InitRock(RenderWindow& window, Texture& tex);   
};

int main()

{

vector<object> objects;

Texture shipTex;

LoadTexture(“data/ship.png”, shipTex);

Object ship;

ship.InitShip(window, shipTex);

objects.push\_back(ship);

Texture rockTex;

LoadTexture(“data/asteroid.png”, rockTex);

Object rock;

size\_t is a custom type, you could make one, think of it as an int, but can hold bigger numbers.

rock.InitRock(window, rockTex);

objects.push\_back(rock);

while(window.isOpen())

{

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

{

objects[i].Update(window.getSize(), elapsed);

objects[i].Render(window);

}  
}

}

Now you need to modify the Update and Render functions so different behaviour runs depending on what type the object is set to.

Initialise needs to call a function for every object type and if a type is missed then assert and stop the program.

Void Object::Init(RenderWindow& window, Texture& tex, ObjectT type\_)

{

switch (type\_)

{

case ObjectT::Ship:

InitShip(window, tex);

break;

case ObjectT::Rock:

InitRock(window, tex);

break;

default:

assert(false);

}

}

Update is similar, but it’s OK to ignore objects other than those that actually need to do something – objects that don’t move don’t update.

Void Object::Update(RenderWindow& window, float elapsed)

{

colliding = false;

switch (type)

{

case ObjectT::Ship:

PlayerControl(window.getSize(), elapsed);

break;

}

}

For now, all objects, regardless of type, render in the same way.

void Object::Render(RenderWindow& window, float elapsed)

{

window.draw(spr);

}

If everything is working you should have a rock in the middle of the screen and a ship you can move around the screen without going off-screen. Both objects are instances of a struct called ‘Object’. As they are both the same type they can both go in the same array (a vector), but the objects act differently because they have different state: one is a ship and the other is a rock.

### Detect when we collide with an asteroid

Use simple circle to circle collision (Pythagoras) to check if you are touching, show debug info about the scene including all the collision radii for the objects, change the colour when a collision is detected.

/\*a2 + b2 = c2 see how close two 2D positions are, see if they are colliding

Precon – pos1+pos2 are 2D positions representing objects moving around

IN: pos1, pos2 – a pair of 2D coordinates

IN: minDist – if they two coordinates are closer or as close as this->collision

RETURNS: true if collision detected

Postcon – two circles are either colliding or not

\*/

bool CircleToCircle(const Vector2f& pos1, const Vector2f& pos2, float minDist)

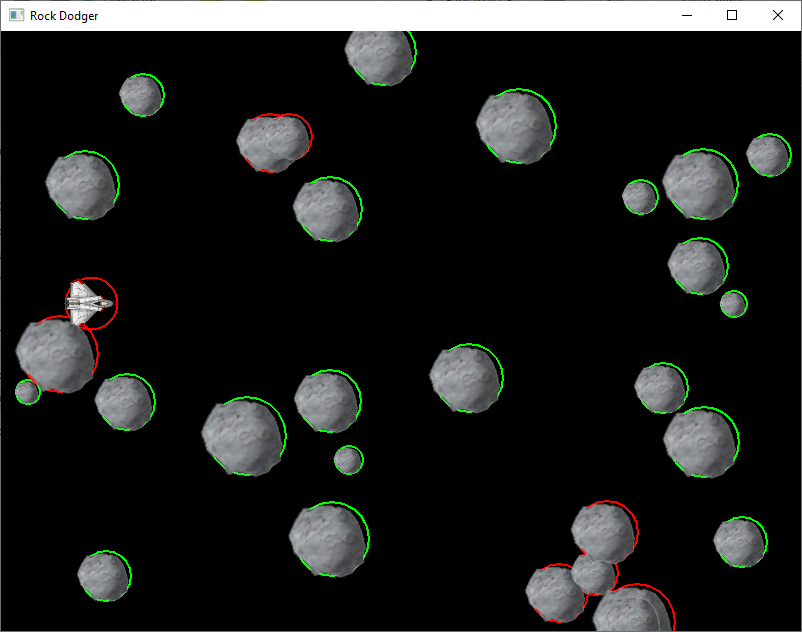
{

float dist = (pos1.x - pos2.x) \* (pos1.x - pos2.x) +

(pos1.y - pos2.y) \* (pos1.y - pos2.y);

dist = sqrtf(dist);

return dist <= minDist;

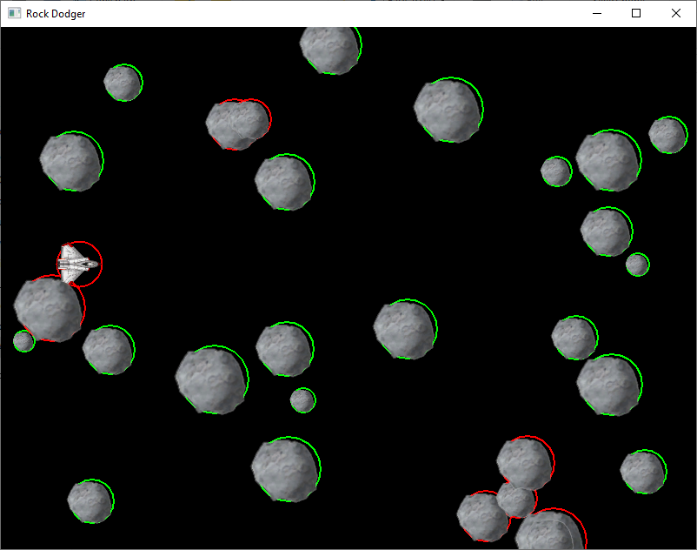


}

Add a new member “int radius” to your Object struct and use that in the collision calculation. How big should radius be though? Add debug functionality to the Object render function so it draws a CircleShape at the radius specified. You can then see if it’s accurate or not. You can’t really “measure” it, what does that even mean? This game isn’t in metres, cm, mm or feet and inches – agreeing a “world scale” with your artists is a very good idea!

Change the colour of the circle depending on whether the object is colliding or not.

### Fill the screen with asteroids

Randomly fill the screen with 30 asteroids and check for collisions with all of them. Some are bound to be overlapping. Green circles mean no collision, red circles mean a collision is detected.

There are a few ways you can do this, for example:

Object rock;

rock.InitRock(window, rockTex);

for(int i=0;i<30;++i){

objects.push\_back(rock);

int x = rand()%1200;

int y = rand()%800;

objects[i].spr.setPosition(x,y);

}

The push\_back command is making a copy of the rock, then resizing the vector array to add it on the end (by allocating a new bigger array internally, then copying everything across and deleting the old array). That’s slow and likely to fragment memory, this is a better way:

Object rock;

rock.InitRock(window, rockTex);

objects.insert(objects.end(), 30, rock);

for(int i=0;i<30;++i){

int x = rand()%1200;

int y = rand()%800;

objects[i].spr.setPosition(x,y);

}

The insert command says, “insert 30 copies of rock at the end of the vector in one go”. This is better than calling push\_back 30 times as the memory inside the vector will be resized and copied just once (faster and less chance of fragmentation). Even better would be to ask the vector how many items it had, rather than hard coding the 30 in the for loop.

Make each asteroid a different size by altering the scale value on the sprite and then multiplying the radius by the scale so the collision code matches the new size.

const Dim2Df ROCK\_RAD{ 10.f,40.f };

void Object::InitRock(RenderWindow& window, Texture& tex)

{

spr.setTexture(tex);

IntRect texR(0, 0, 96, 96);

spr.setTextureRect(texR);

spr.setOrigin(texR.width / 2.f, texR.height / 2.f);

radius = GC::ROCK\_RAD.x + (float)(rand() % (int)GC::ROCK\_RAD.y);

float scale = 0.75f \* (radius / 25.f); //fiddly to get it quite right

spr.setScale(scale, scale);

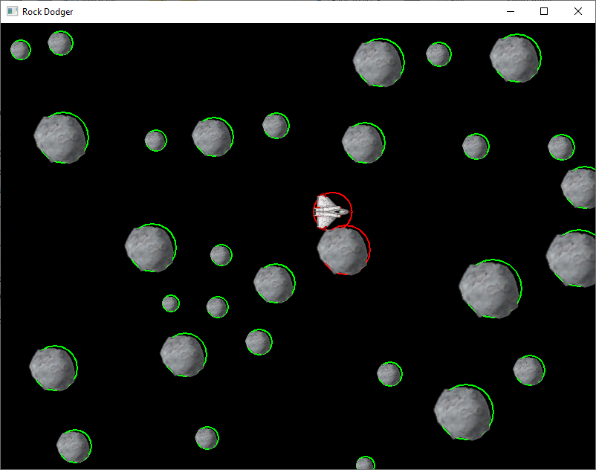
type = ObjectT::Rock;

}

As you move the ship around, the collision system should still work and look correct.

### Avoid overlap

It looks a bit pants when they overlap so use the collision code to stop that happening. First, add some new state to the Object struct ‘bool active = false;’. This means we can have many rocks, but only **active** ones are rendered and have collision checks done.

Pre-condition – no rocks are marked active

For each object in the container

If it’s a rock

Pretend it’s bigger than it really is

Set it active

Do

Set a random on-screen position

For every item in objects

If it isn’t this object and it is active

Check if it collides with us

While(colliding and we’ve tried < 10 times)

Note: this is a heuristic algorithm in that the solution isn’t necessarily optimal, it gives results that are good enough. In the worst case it won’t find a clear position within 10 attempts and you will get an object overlapping still. The chance of this gets worse the bigger the rocks are and the more of them there are. Notice how we temporarily pretend the new rock is much bigger than it really is when checking for collisions, this is because we need room around all the rocks to fly the ship (we want big gaps).

namespace GC

{

const float ROCK\_MIN\_DIST = 2.15f; //stop them getting too close

const int NUM\_ROCKS = 50; //how many to place

const int PLACE\_TRIES = 10; //how many times to try and place

}

void Game::PlaceExistingRocks(RenderWindow& window)

{

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

{

if (objects[i].type == Object::ObjectT::Rock)

{

Object& rock = objects[i];

rock.radius \*= GC::ROCK\_MIN\_DIST;

rock.active = true;

int tries = 0;

do {

tries++;

float x = (float)(rand() % window.getSize().x);

float y = (float)(rand() % window.getSize().y);

rock.spr.setPosition(x, y);

} while (tries < GC::PLACE\_TRIES &&

IsColliding(rock, objects));

rock.radius \*= 1 / GC::ROCK\_MIN\_DIST;

}

}

}

There’s a solution: [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t11\_collide.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t11_collide.git)

### Spawn asteroids in off screen and fly them past

If you still have the debug collision circles visible then you can see when something hits your ship. This should now feel like a game, try and dodge the rocks 😊

1. However, it would be even more game-like if the rocks moved.

Get sprite position

Move left at a fixed speed

If the object has gone off screen completely then

Set the object not active so it stops rendering and can be re-used later

void Object::MoveRock(float elapsed)

{

const Vector2f& pos = spr.getPosition();

float x = pos.x - GC::ROCK\_SPEED \* elapsed;

if (x < -spr.getGlobalBounds().width / 2.f)

active = false;

spr.setPosition(x, pos.y);

}

Note the use of spr.getGlobalBounds(), this returns the left / top / width / height of the sprite, but it includes any scaling/rotating that might be applied. It is the actual on-screen dimensions of the sprite.

1. Now they are moving, spawn rocks in off-screen to the right, but make sure they don’t overlap anything else.

While we haven’t found an inactive rock

Get the next object

Check if it is a rock type and if it’s active

If we found one

Set it active

Increase its radius so we definitely have room to fly around it

Get a random height that is on-screen

Set the rock’s position to his height and just off screen to the right

Check if it collides with any other **active** object

If it does then

Set the object to be inactive and forget it

Set the rock’s radius back to the correct value

bool SpawnRock(RenderWindow& window, vector<Object>& objects, float extraClearance)

{

//find a !active rock we can use again

size\_t idx = 0;

bool found = false;

while (idx < objects.size() && !found)

{

Object& obj = objects[idx];

if (!obj.active && obj.type==Object::ObjectT::Rock)

found = true;

else

++idx;

}

if (found)

{

//place it randomly just off screen

Object& obj = objects[idx];

obj.active = true;

obj.radius += extraClearance;

FloatRect r = obj.spr.getGlobalBounds();

float y = (r.height/2.f);

y += (rand() % (int)(window.getSize().y - r.height));

obj.spr.setPosition(window.getSize().x + r.width, y);

if (IsColliding(obj, objects))

{

//if it’s colliding with something then forget it

found = false;

obj.active = false;

}

obj.radius -= extraClearance;

}

return found;

}

This is not a heuristic as it will perform reliably, there is either room for the new rock or there isn’t, but as all rocks move left at the same speed, as long you keep calling this function then it will always succeed eventually and predictably.

Note the extraClearance parameter, this is handled slightly differently than spacing the rocks previously, if we use the radius of the ship plus a bit, then we can be very precise about how much room the player has. The “a bit” value could control the game’s difficulty, getting smaller as time goes on perhaps?

Solution available: [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t11\_dodge.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t11_dodge.git)

### Make the ship feel juicy

Currently the ship’s motion doesn’t feel natural, it’s jerky and instantaneous, like the ship has no weight. When the player lets go of the controls it should take a short time for the ship to slow down. We can fake that by giving the ship velocity that decays over time:

if(!user\_input)

ship.vel \*= 0.95f;

This is very hacky as the rate of decay is linked to the framerate (how fast update is called). A better version takes account of time to ensure it is always the same feel. Alternatively we could make the game update at a fixed rate 30fps perhaps. Anyway, here’s a way to time it properly:

/\*

reduce a vector by a certain percentage over a certain time

currentVel - the vector you need to make a reduced version of

pcnt - at the end of timeInterval it should have reduced by this percentage (e.g. 10%=0.1)

timeInterval - how many seconds for the pcnt reduction to complete

dTimeS - actual elapsed time e.g. 10% after 1sec, so if dTimeS=0.1s, then the reduction will be 1%

\*/

Vector2f Decay(Vector2f& currentVal, float pcnt, float timeInterval, float dTimeS)

{

float mod = 1.0f - pcnt \* (dTimeS / timeInterval);

Vector2f alpha( currentVal.x \* mod, currentVal.y \* mod);

return alpha;

}

Going further we could implement the whole thing as a physics simulation, the ship becomes a point mass subject to thrust forces, terminal velocity and friction. Consult your favourite physics book for ideas.

Solution available: [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t11\_dodge.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t11_dodge.git)

### Bullet

Add a third object type ‘Bullet’ and let the player fire off a bullet to damage an asteroid. The sprite I found has two frames, if you alternate them at a fixed time interval (animation), it looks like the exhaust flame is pulsing.



void Object::InitBullet(RenderWindow& window, Texture& tex)

{

spr.setTexture(tex);

IntRect texR(0, 0, 32, 32);

spr.setTextureRect(texR);

spr.setOrigin(texR.width / 2.f, texR.height / 2.f);

radius = 5.f;

float scale = 0.5f;

spr.setScale(scale, scale);

active = false;

type = ObjectT::Bullet;

health = 0;

}

Initialising the game now gets quite complicated, we want one ship, 50 rocks and 50 bullets, with only the ship active initially:

void Game::Init(sf::RenderWindow & window) {

LoadTexture("data/ship.png", texShip);

LoadTexture("data/asteroid.png", texRock);

LoadTexture("data/missile-01.png", texBullet);

objects.clear();

Object obj;

objects.insert(objects.begin(), GC::NUM\_ROCKS+GC::NUM\_BULLETS+1, obj);

size\_t idx = 0;

objects[idx++].Init(window, texShip, Object::ObjectT::Ship,\*this);

for (idx; idx < (GC::NUM\_ROCKS+1); ++idx)

objects[idx].Init(window, texRock, Object::ObjectT::Rock,\*this);

for (idx; idx < objects.size(); ++idx)

objects[idx].Init(window, texBullet, Object::ObjectT::Bullet,\*this);

}

If you add health to our Object struct then in collision detection code, if a bullet is hitting a rock then just reduce the health of the rock and get rid of the bullet (active=false). Rocks could have health proportional to their radius, so a bullet does one damage and a big rock has five health, while a tiny rock has one health. Your ship could have three health.

void Object::Hit(Object& other)

{

switch (type)

{

case ObjectT::Ship:

if (other.type == ObjectT::Rock)

{

TakeDamage(1);

other.TakeDamage(999);

}

break;

case ObjectT::Bullet:

if (other.type == ObjectT::Rock)

{

TakeDamage(1);

other.TakeDamage(1);

}

break;

case ObjectT::Rock:

break;

default:

assert(false);

}

}

void Object::TakeDamage(int amount)

{

health -= amount;

if (health <= 0)

active = false;

}

Then alter the update function to handle rocks and pass in whether the fire button is pressed or not.

void Object::Update(RenderWindow& window, float elapsed, bool fire)

{

if (active)

{

colliding = false;

switch (type)

{

case ObjectT::Ship:

PlayerControl(window.getSize(), elapsed, fire);

break;

case ObjectT::Rock:

MoveRock(elapsed);

break;

case ObjectT::Bullet:

MoveBullet(window.getSize(), elapsed);

break;

}

}

}

It’s like a game!!!!!!! ☺

Solution available: [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t11\_shoot.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t11_shoot.git)

|  |
| --- |
| ASSESSED HOMEWORK Can you take our last task, which was a shootemup with bullets and asteroids, then add the scrolling background mountain scene we worked on earlier?Background parallax layers scroll in from the right, the mountain scene. [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t08\_spritescrollsol.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t08_spritescrollsol.git) Player ship can move around using the arrow keys but remains on screen. Asteroids randomly spawn in off screen to the right, without overlapping and then fly straight across. [**https://fezztah@bitbucket.org/fop4g\_sysmod4g/t11\_shoot.git**](https://fezztah@bitbucket.org/fop4g_sysmod4g/t11_shoot.git) Keep a copy in your GiT repository, add a screen grab of your commit log to your design doc. Use the design doc template on BB. Competencies: 1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8 or 1.9, 2.1,2.2,2.4,2.6,2.7,2.8 |

|  |
| --- |
|  |