INFO0004-2: Project 3
Drunk Run 2.5D

By groups of 2

April 23, 2021

1 Introduction

Cymi Saril is an INFO0004 student who likes to party and has been very miserable during this last year of pandemic. We are not here to judge, but Cymi has decided to go to the Parc de la Beuverie, in the center of Liège, with a bunch of friends. It is now time to go home, but for Cymi, a touch intoxicated, that won't be that easy...

The way to Cymi's home from the park is just a long, straight path. But at this time of night, it is full of rubbish (people should really take their shit home!), other people (just standing there, drinking and talking), as well as barriers put there by yobs hoping they would slow down the police should they give chase.

A lot of the rubbish is just small stuff, like cups and cans which, while very unsightly, will pose no danger for Cymi. But disaster will strike if Cymi trips on any bigger item and falls over. The path also has a kerb on both sides, that will trip Cymi, so better stay away from it!

In this project, you will be building a simple game in C++17, using the SDL 2.0.14 library (Simple Directmedia Layer). This library provides easy windowing and user input, timing and graphics rendering facilities.

In this game, the player helps Cymi *navigate* the dangers on the path.

Your game must render at 30 frames (images) per second.

2 The game view

The player sees what Cymi sees (see figure 1a).

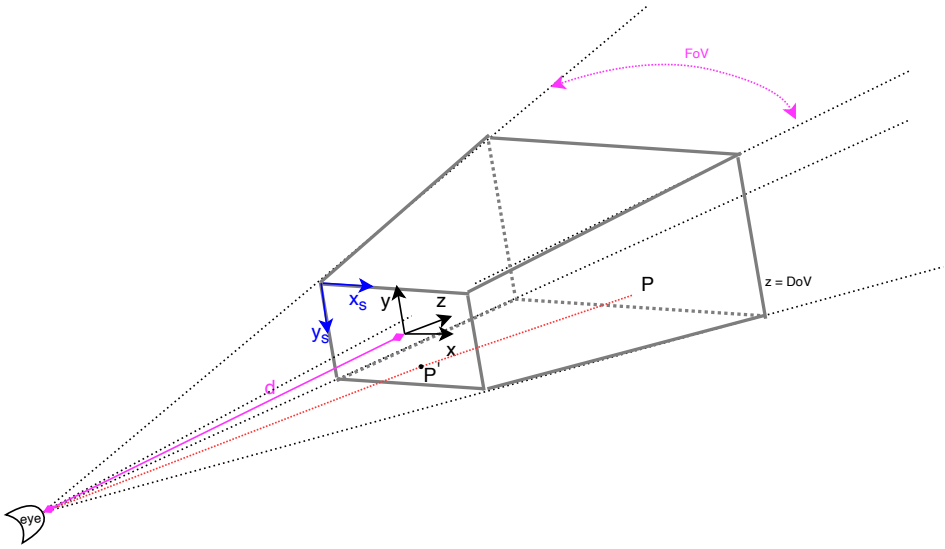
In this game, to simplify, all obstacles are rectangular cuboids (a.k.a boxes), and the scene is rendered as a single-point perspective.

In single-point perspective (see figure 1), the front side (and back side) of a box is *always* rendered as a perfect rectangle, while lines that are orthogonal to these sides are rendered as converging towards a vanishing point in the middle of the screen (figure 1b).

This is a simplification on reality: in the real world, the front side of a box that is not perfectly orthogonal to the line of sight, and perfectly centered with the eye, will be seen as a trapezoid (a *distorted* rectangle). However, the effect obtained by the one-point perspective is good at creating a sense of depth, and this is why we call it 2.5D, as it is not quite 3D.

(a) Game view.

(b) Box mesh.



(c) Computing perspectives.

Figure 1: ***1-point perspective.***

2.1 Computing perspectives

Computing perspectives is actually quite easy. All we need to do is *simulate* what light does to form an image in an eye: in figure 1c, the perspective P' of point P is the intersection of a ray of light cast between P and the eye, and the *screen* where the image is formed.

We could just tell you that and let you figure out the math, but since we are nice (if we say so ourselves, we must be!), we are going to give you the formulas:

Let the origin of the 3D referential be the middle of the screen, oriented as in figure 1c and the eye be at $(0, 0, -d)$, the projection of $P(x, y, z)$ is

$$P'(xd/(z+d), yd/(z+d), 0)$$

Note that this approach does not work for points situated in front of the screen ($z < 0$), so such points should not be used in perspective computations.

For this project, we will consider that the eye has a field-of-view (FoV) of $60°$ (horizontally) (see figure 1c) and an aspect ratio of 4/3 (the width of the image is 4/3 times its height).

Finally, points beyond the depth-of-view ($z >$ DoV) are considered too far and should not be rendered either. In general, points outside the solid truncated pyramid in figure 1c will not be visible to the eye because outside of the field-of-view.

Hint: Similar right triangles theory will be your friends in determining which points are within the field-of-view.

2.2 Rendering

Note that if you use the perspective coordinates of a point to display that point, the 3D coordinate units must then be in pixels, which are very awkward to manipulate. Furthermore, changing the size of the display window would then *scale* the view, which is undesirable.

We therefore suggest that you set the 3D coordinate units to something relatable in the real world (like centimetres), and then scale the computed perspective coordinates of points to fit the window size. You should also keep in mind that the SDL2 window coordinate system is oriented as the x_s and y_s axes in figure 1c.

In 3D, determining which surfaces are (partly) visible is usually not obvious, but in single-point perspective, it is very easy: if you render boxes in the reverse order of the depth (z-coordinate) of their front face, hidden-surface determination will be correct! In other words, start with the boxes whose front faces are the furthest away.

Finally, to simplify, you can render boxes as meshes (figure 1b), but for a 2 point bonus, you can choose to render solid boxes (figure 1a).

Bonus hint 1: Determining which faces of a solid box are visible is very easy.

Bonus hint 2: SDL2 has a primitive to render painted rectangles. For anything that is not a rectangle, paint it as you would paint a wall, by making lines of colour (like you do with a paintbrush).

Bonus hint 3: Beware of numerical instabilities.

Bonus hint 4: To simplify, you can assume that boxes exit the field of view when their front face reaches a depth of zero.

3 The gameplay

Cymi will just be walking at a constant speed straight ahead, encountering whatever is on the path, trying to avoid the bigger items in order not to trip. Cymi will be controlled by two keys, say `L` to go left and `M` to go right. To simplify, any lateral movement can just be a little *side jump* (instantaneous shift).

The problem is that Cymi is quite drunk, so lateral movements will occur randomly, outside of Cymi's control too. A simple staggering model might be to generate a side shift equivalent to between 1 and 6 lateral movements, in either direction, randomly, on average every 2 seconds.

As an example, the game in figure 1a has 4 different types of obstacles:

1. Rubbish (gray), whose size is 15x7x7 cm^3.
2. Crates (yellow), 45x18x30 cm^3.
3. Barriers (red), 120x90x20 cm^3.
4. People (blue), 40x180x40 cm^3.

Each obstacle is associated with a colour, which is used to draw the edges of the boxes. Then, the red, green and blue values of that colour are divided by two to draw the sides of the boxes, simulating a depth effect.

Regarding the parameters that we previously discussed, the path is 4 m wide and Cymi's eyes are 1.7 m above the ground, 20 cm behind the image screen. The DoV is 150 m.

Again, to simplify, you can model Cymi as a box with a single lateral movement being equivalent to 5 units.

All these parameters, as well as the distribution of obstacles (in our example, there are 30% of dangerous obstacles and 70% of rubbish) throughout the path can be modified to adjust the game play and/or difficulty of the game—but to keep things simple, just stick to one set of parameters in a game.

4 Scoring

The score is simply the distance, along the z-axis, that Cymi has walked.

5 Remarks

- **Time keeping:** time keeping in the game does not need to be super precise. At 30 frames per second, each frame is just over 33 ms. This is plenty a resolution for your time keeping. Also, 33 ms is plenty of time to compute a frame of the game in C++, so you can just assume that the frames arrive bang on time, and you can use these frames as a time reference (you do not need a wall clock). Any clock drift from this assumption will not cause any problem.
- **Starting the game:** at the start of the game, you should give the player a little bit of clear path, so there is time to react to the first coming obstacles.
- **Be careful:** the game in itself is very simple, but the graphical sub-system will require some work.
- It is perfectly OK to just display "Game Over" for a few seconds, then exit the application when Cymi has tripped. Take a look at `SDL_ttf 2.0` for font rendering.

5.1 Readability

Your code must be readable:

- Make the organisation of your code as obvious as possible.
- Use descriptive names.
- Complement your self-documenting code with comments, where appropriate.
- Choose a coding convention, and stick to it. *Consistency* is key.

5.2 Robustness

Your code must be robust. The `const` keyword must be used correctly, sensitive variables must be correctly protected, memory must be managed appropriately, the program must run to completion **without crash**.

5.3 Warnings

Your code must compile **without error or warning** with `g++ -std=c++17 -Wall` on the provided container (`cffs/oopp`). However, we advise you to check your code also with `clang++ -std=c++17 -Wall -Wextra -pedantic`, and tools such as valgrind or cppcheck.

5.4 Object-oriented approach

Remember this is an *object-oriented* course. Try to apply the object-oriented concepts where it makes sense.

5.5 Makefile

You **must** provide a `Makefile` that can build your game in the given container. That `Makefile` should support separate, incremental compilation (*i.e.* if one file is modified, only that file and files depending on it should be recompiled).

5.6 Evaluation

Your code will be evaluated based on all the above criteria. Failure to comply with any of the points mentioned **bold face** can result in a mark of zero!

6 Submission

Projects must be submitted through the submission platform before **May 16$^{\text{th}}$, 23:59 CEST**. Late submissions will be accepted, but will receive a penalty of $2^n - 1$ points (/20), where n is the number of days after the deadline (each day start counting as a full day).

You will submit a `<GROUPID>.tar.xz` archive of a `<GROUPID>` folder containing your C++ source code, and all other files needed for your code to run, and a `Makefile` to compile it (with a simple `make`), where `<GROUPID>` is your group ID on the submission platform.

This archive must also contain a report.pdf file, no longer than 5 pages, that describes the design and architecture of your code.

As it would be hard to test your game automatically, the submission platform will not do basic checks on your submission. You can submit multiple times.