Week 05

Spritesheets, Animation, and Text

3 Wyvern Moon, Imperial Year MMXXIV

Song of the day: **果実 (Fruit)** by Akasaki (2023)

Sections

Part 1: *A Bit of Review of Textures*

As you may have realised while completing your first project, having more than one texture can quickly get out of hand. Moreover, if we want to *animate* our objects on-screen, we would have to create a different texture ID for each individual frame of animation. There is, naturally, a correct way of doing this, and we'll get into it today. Recall that textures follow UV-coordinates:

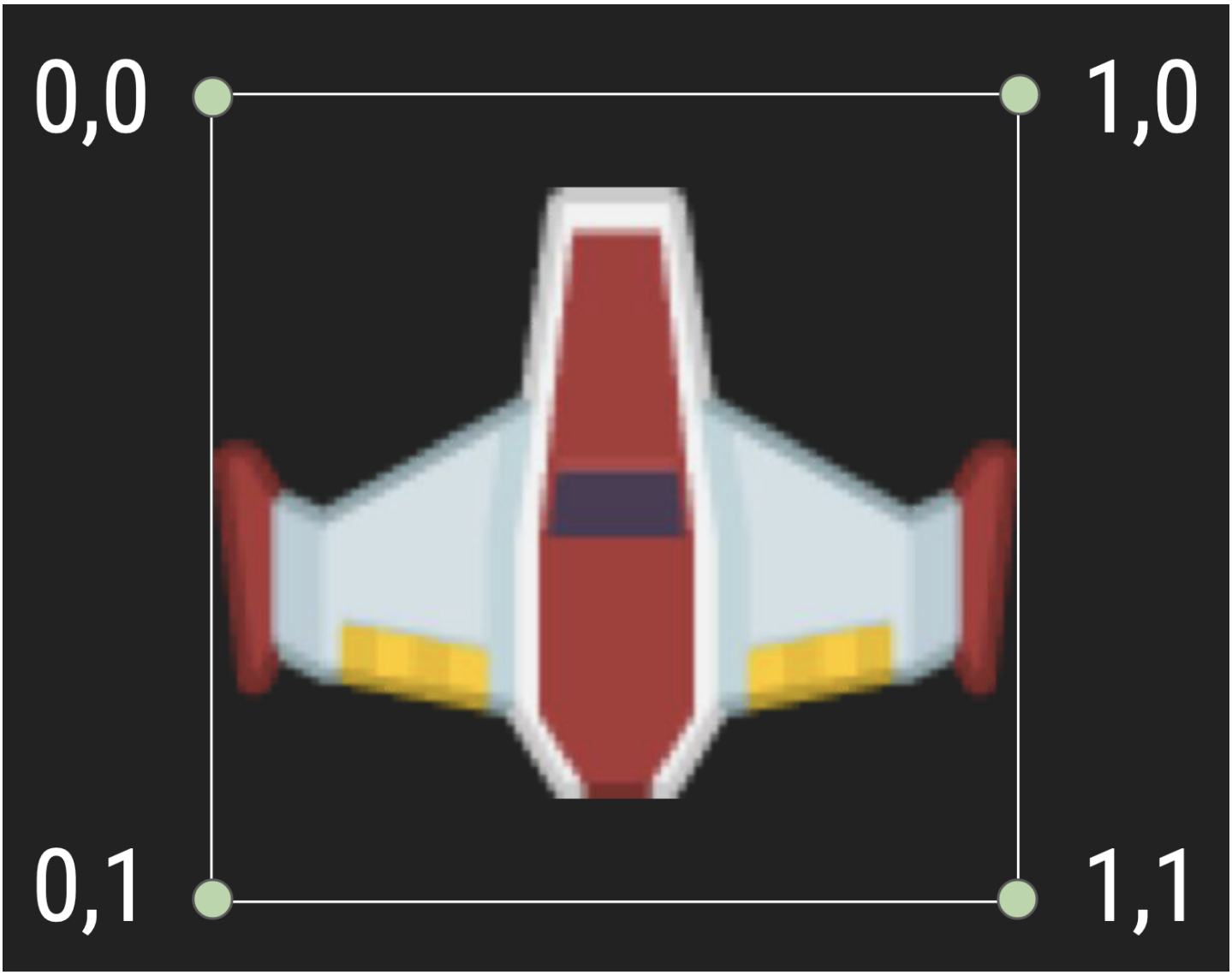
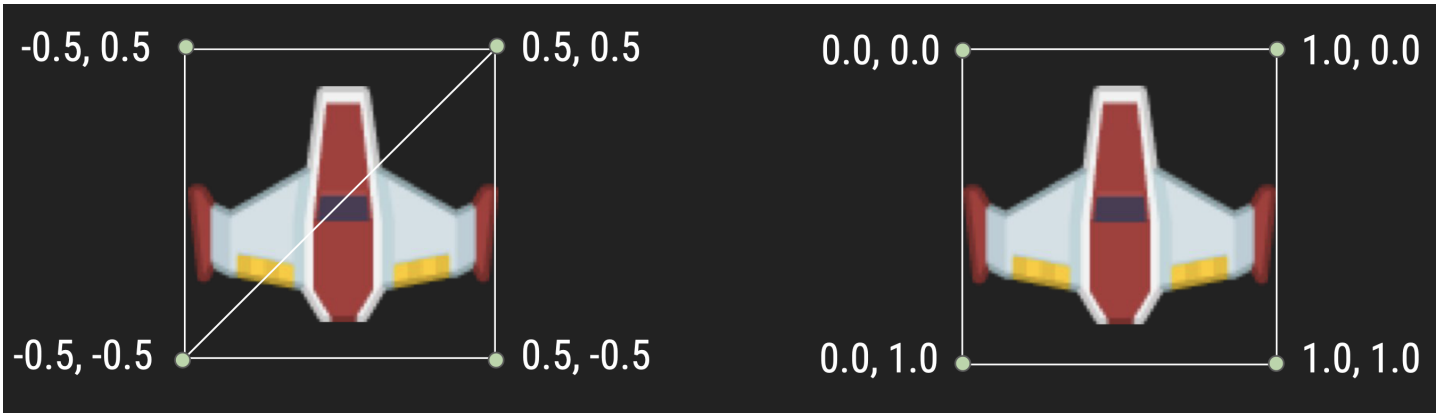


Figure 1: UV-coordinates superimposed over a texture covering two triangles.

Recall, too that the way we translate these UV-coordinates to our world's XY-coordinates is via two sets of vertices:




```
float vertices[]  = {-0.5, -0.5, 0.5, -0.5, 0.5, 0.5, -0.5, -0.5, 0.5, 0.5,
-0.5, 0.5};
float tex_coords[] = { 0.0,  1.0, 1.0,  1.0, 1.0, 0.0,  0.0,  1.0, 1.0, 0.0,
0.0, 0.0};
```

Figure 2 and **Code Block 1**: UV-coordinates to XY-coordinates.

Now, something that some of you hinted at recently is the possibility of changing the values inside our `tex_coords` array; especially when our sprites are not necessarily a square, the ability to isolate parts of it is especially helpful.

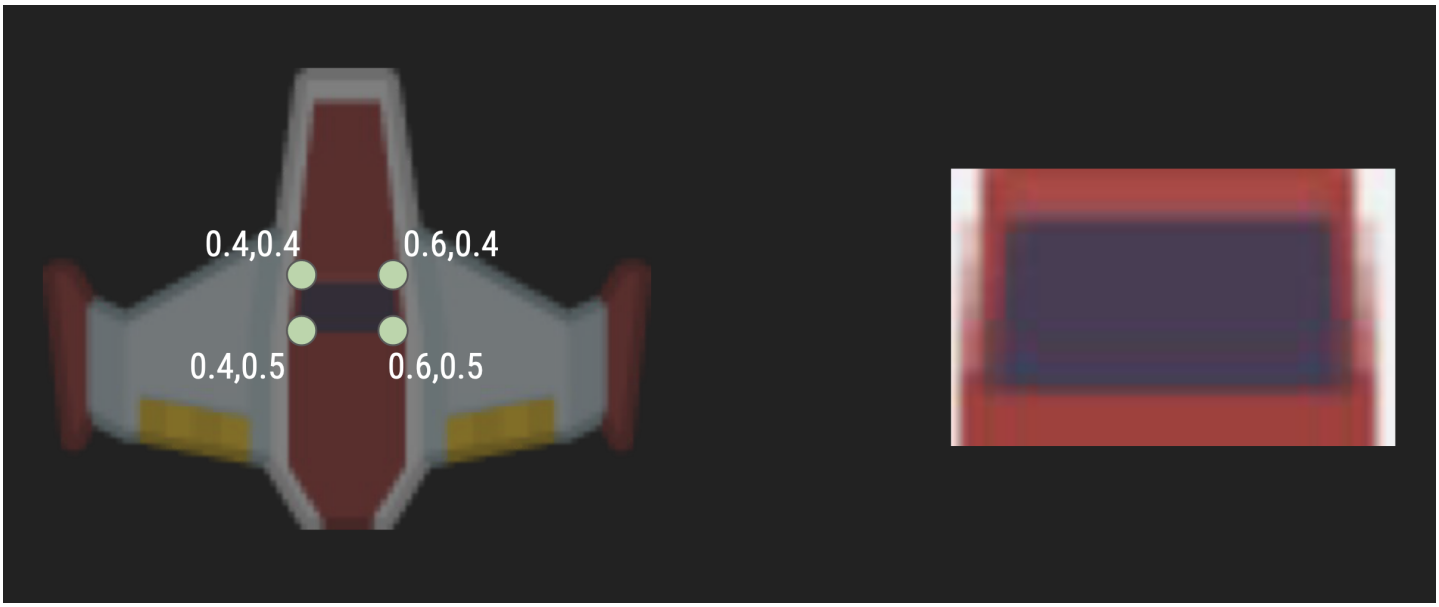


Figure 3: Instead of only using values [0.0, 1.0], we can isolate certain parts of the texture.

There are bigger implication to our ability to "slice" a texture, and we've reached a point where doing so will allow for a tremendous amount of flexibility.

Let's look at a couple of instances where changing the values in `tex_coords` might affect our game, and how we can handle them.

Part 2: *Texture Wrap Mode*

If our UV- and XY-coordinates match one-to-one, we have it pretty simple. After all, there's no amount of "compensation" we have to do; our corners match perfectly in both coordinate systems:

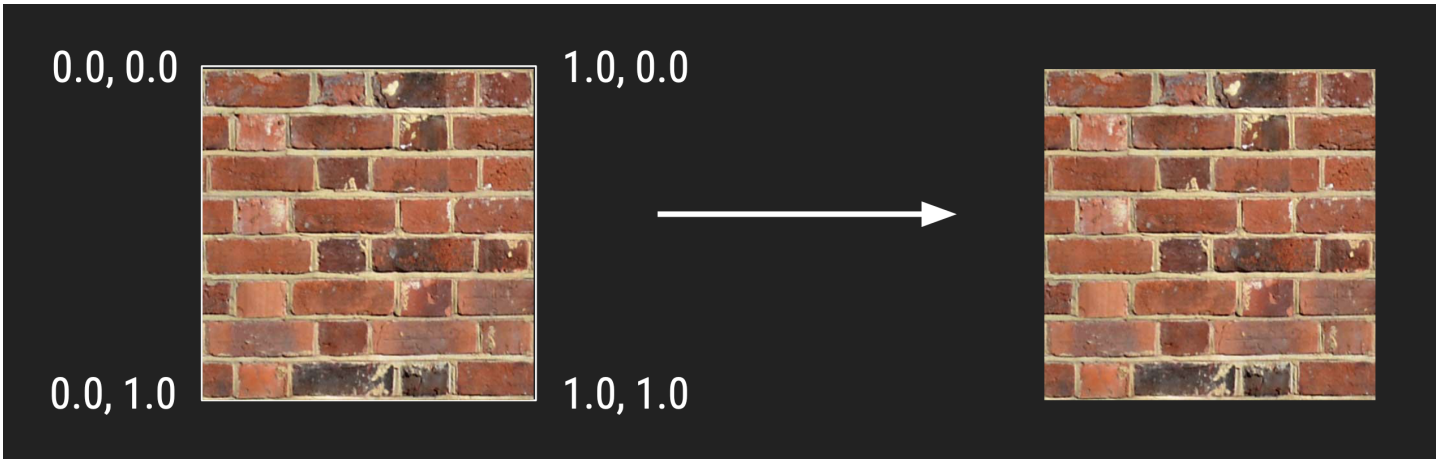
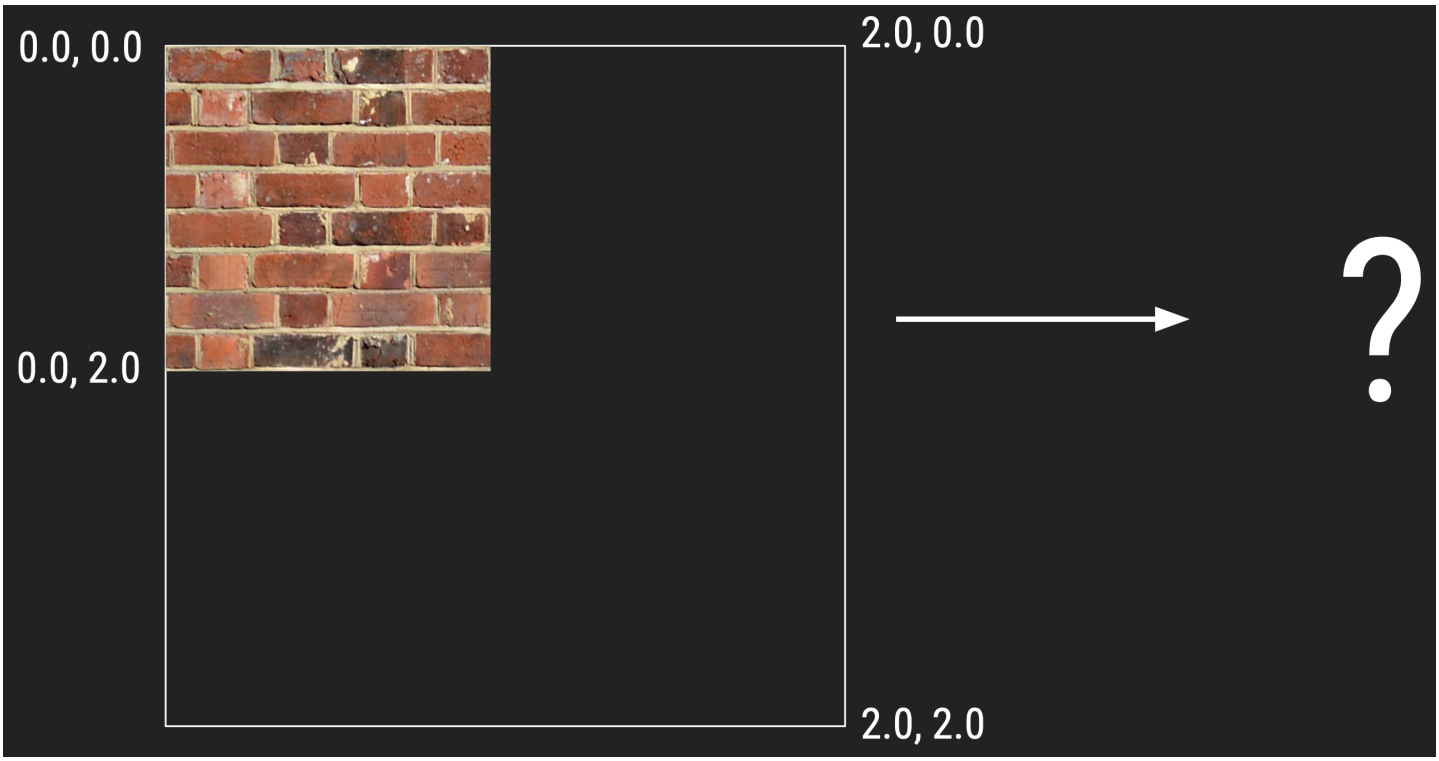


Figure 4: Wouldn't it be nice if the world was ever this neat?

This set of bricks looks perfect for a Mario-style worldbuilder. In this case, creating separate objects for *every single brick block* in the scene would be both exhausting and completely ineffective. So what do we do in

situations like these?




```
float vertices[] = { -0.5, -0.5, 0.5, -0.5, 0.5, 0.5, -0.5, -0.5, 0.5, 0.5,
-0.5, 0.5 };
float tex_coords[] = { 0.0, 2.0, 2.0, 2.0, 2.0, 0.0, 0.0, 2.0, 2.0, 0.0, 0.0,
0.0 };
```

Figure 5 and **Code Block 2**: What if our object was a 2x2 set of brick blocks?

The answer to our plight in this case is actually super simple, and it comes in the form of **texture wrap modes**. Of these, four exist in OpenGL:

Figure 6: OpenGL wrapping modes. (**Source**)

Quite simply, this is a matter of adding two more `glTexParameteri()` calls inside our `load_texture()` function:

```
// Setting our texture wrapping modes
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT); // the last
argument can change depending on what you are looking for
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Code Block 3: Just like with filter modes, this is all that is necessary for wrapping.

Part 3: *Texture Atlases and Sprite Sheets*

We've just looked at how to handle textures that are smaller than our objects, but how are we handle objects that are smaller than our textures? It turns out that this is exactly how we will be doing graphics for a number of situations—perhaps for all of them. Introducing the texture atlas, or as it is more commonly known, the sprite sheet:

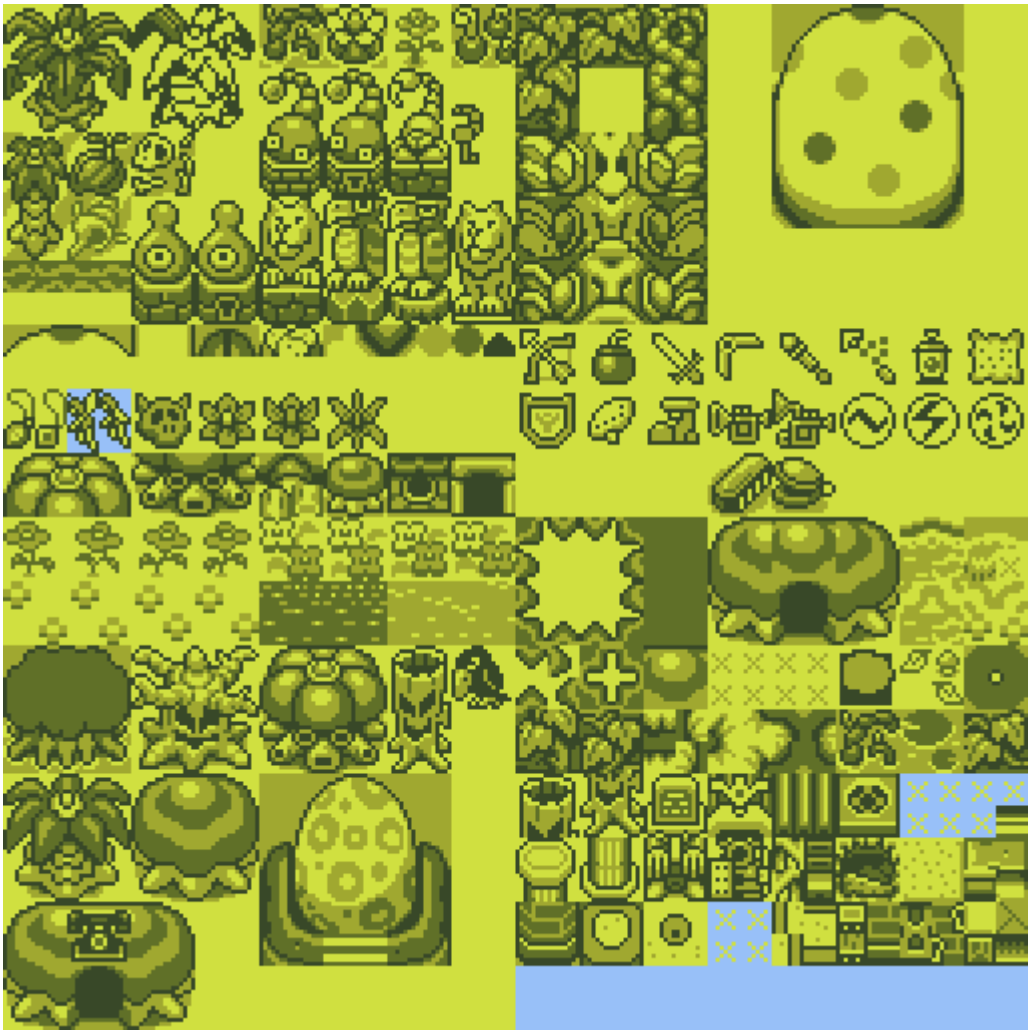


Figure 7: *Link's Awakening*'s sprite sheet.

What you see in figure 7 is a single texture. If we were to apply it to our current object, we would indeed get the whole texture onto it, but that is not the goal here. Our goal is **isolate each individual portion of the image and make it its own texture**:

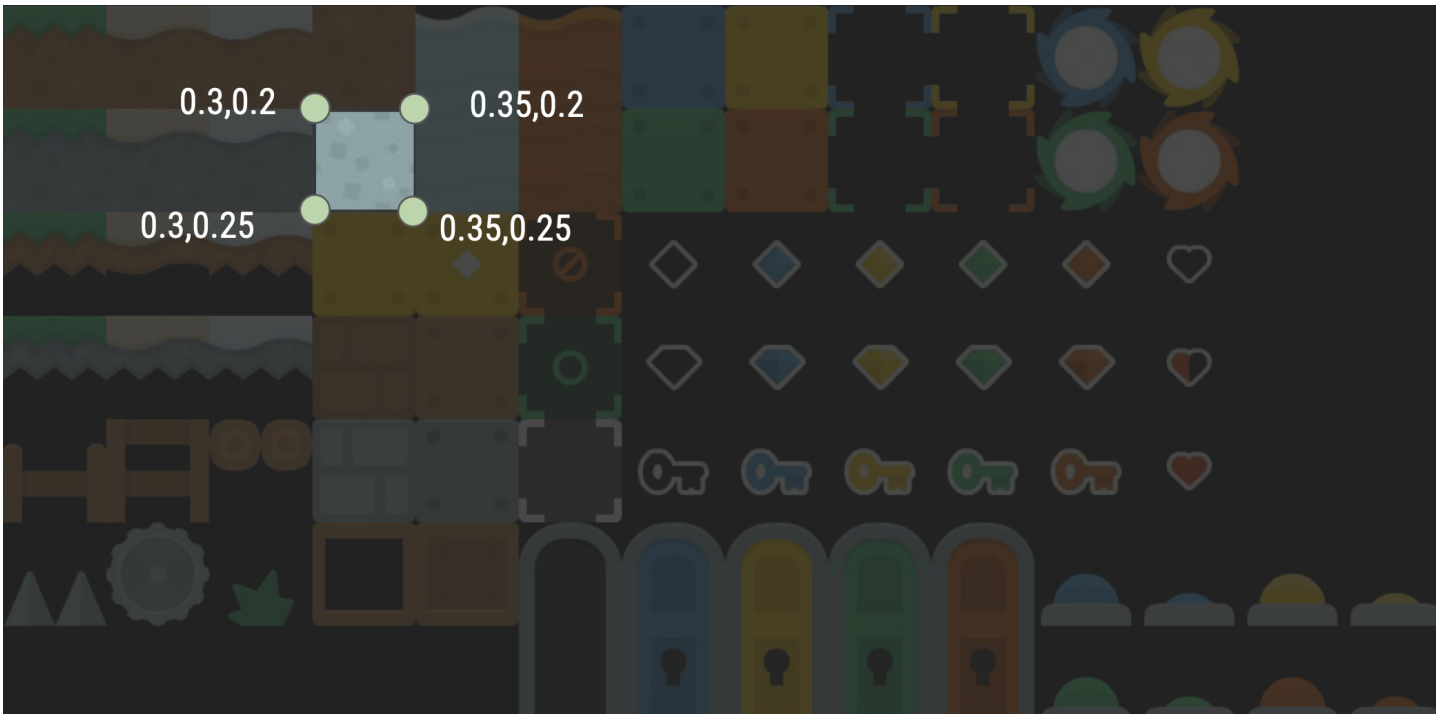


Figure 8: Sprite sheet with texture coordinates superimposed on it.

This can be applied to something that is perhaps more critical to game development—fonts:

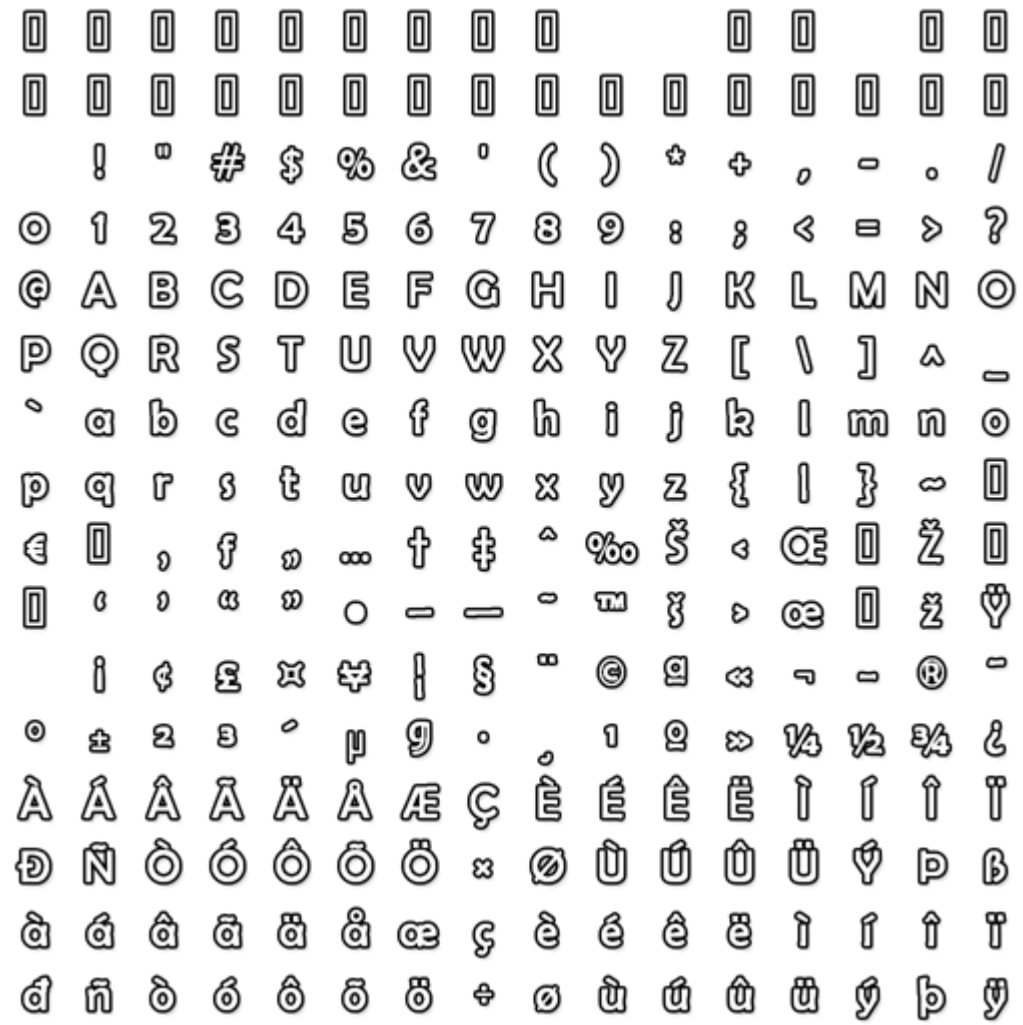


Figure 9: A sprite sheet with sprites that just happen to be characters.

When picking sprite sheets, it's important to keep that having them be **monospaced**. This simplifies the process significantly, as we'll see soon. In other words, don't pick a spritesheet that looks like this:

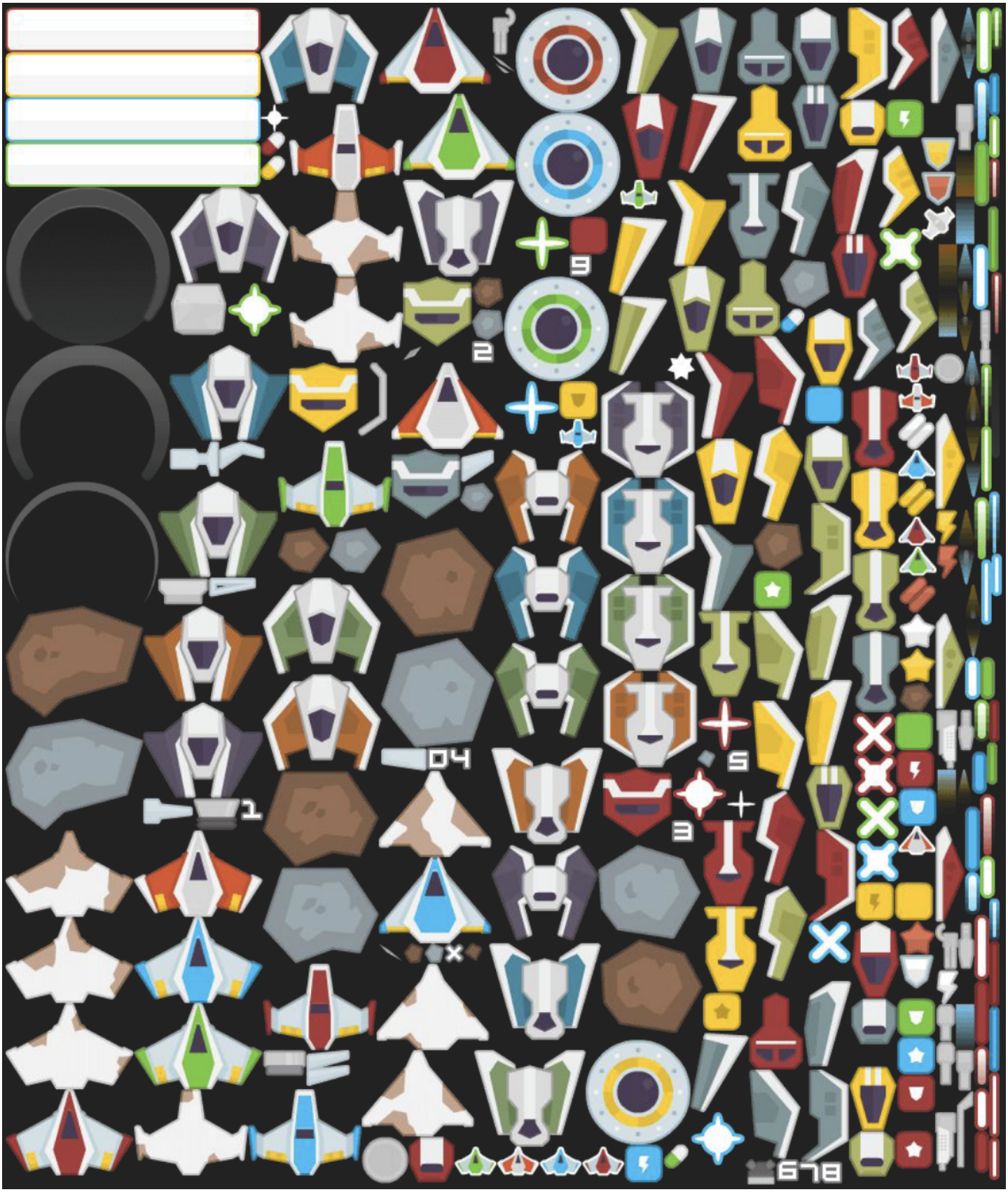
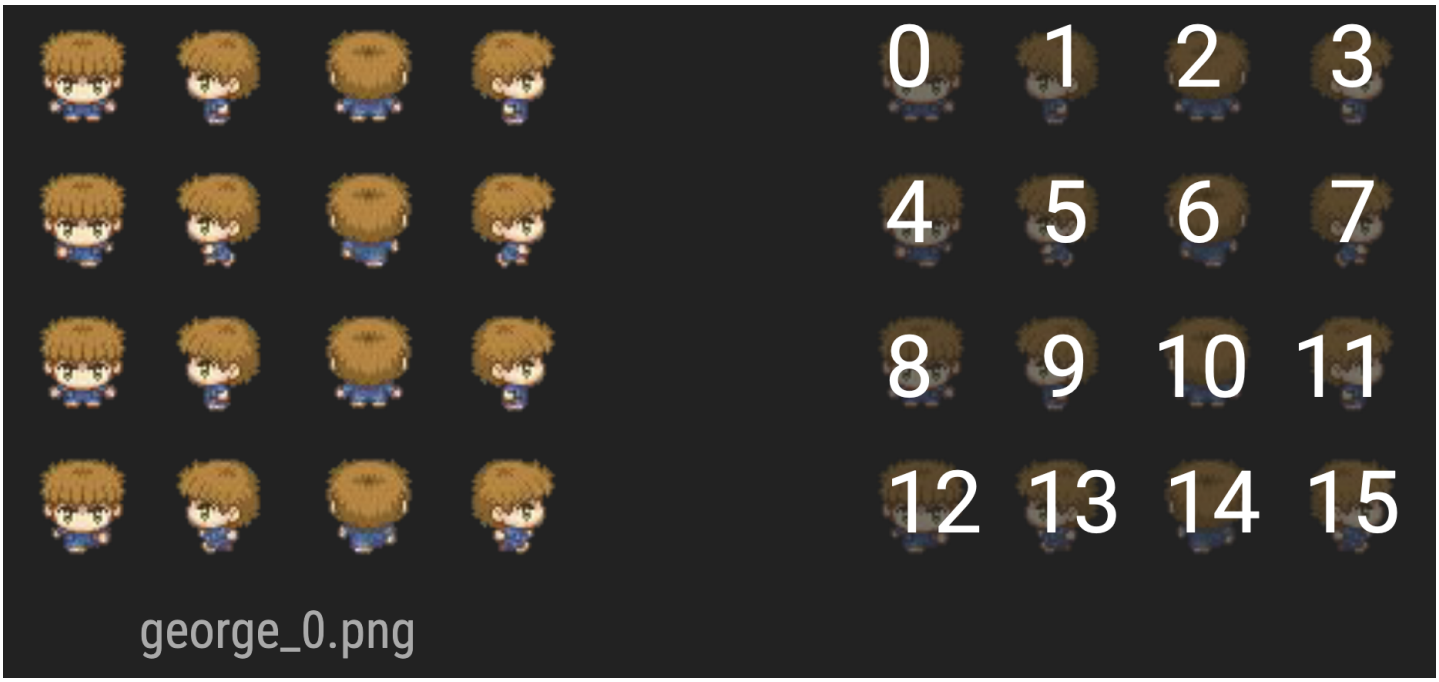


Figure 10: It's basically impossible to split this into a uniform grid.

The reason why grids over sprite sheets are so important is because it makes it easy to programmatically parse and use:



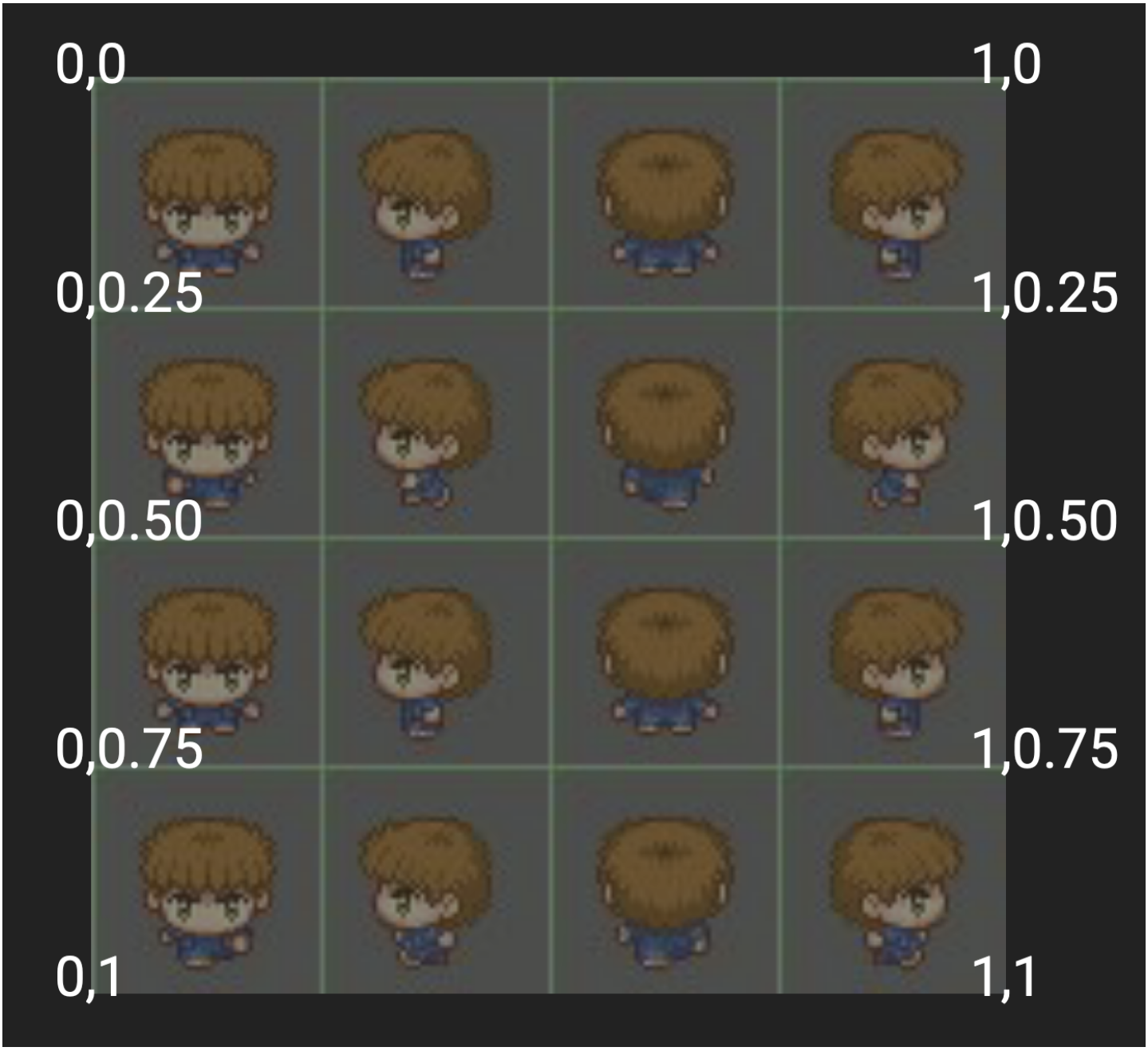


Figure 11 and 12: Different animation frames as a sprite sheet.

So, how do we program this? Well, using the sprite sheet example in figures 11 and 12, the way we would pick the desired frame would be as follows:

```
void draw_sprite_from_texture_atlas(ShaderProgram *program, GLuint texture_id,
int index, int rows, int cols)
{
    // Step 1: Calculate the UV location of the indexed frame
    float u_coord = (float) (index % cols) / (float) cols;
    float v_coord = (float) (index / cols) / (float) rows;

    // Step 2: Calculate its UV size
    float width = 1.0f / (float) cols;
    float height = 1.0f / (float) rows;

    // Step 3: Just as we have done before, match the texture coordinates to
the vertices
    float tex_coords[] =
    {
        u_coord, v_coord + height, u_coord + width, v_coord + height, u_coord +
width, v_coord,
        u_coord, v_coord + height, u_coord + width, v_coord, u_coord, v_coord
    };

    float vertices[] =
    {
        -0.5, -0.5, 0.5, -0.5,  0.5, 0.5,
        -0.5, -0.5, 0.5,  0.5, -0.5, 0.5
    };

    // Step 4: And render
    glBindTexture(GL_TEXTURE_2D, texture_id);

    glVertexAttribPointer(program.get_position_attribute(), 2, GL_FLOAT, false,
0, vertices);
    glEnableVertexAttribArray(program.get_position_attribute());

    glVertexAttribPointer(program.get_tex_coordinate_attribute(), 2, GL_FLOAT,
false, 0, tex_coords);
    glEnableVertexAttribArray(program.get_tex_coordinate_attribute());

    glDrawArrays(GL_TRIANGLES, 0, 6);

    glDisableVertexAttribArray(program.get_position_attribute());
    glDisableVertexAttribArray(program.get_tex_coordinate_attribute());
}
```

Code Block 4: Basically what we've been doing so far, but a bit more specialised.

Using this function, our render function becomes much shorter and neater:

```
constexpr int SPRITESHEET_DIMENSIONS = 4; // this depends on the spritesheet,
obv
```

```
void render()
{
    glClear(GL_COLOR_BUFFER_BIT);

    g_shader_program.set_model_matrix(model_matrix);

    // Here, the 0th sprite is being rendered. Hence, the 0
    draw_sprite_from_texture_atlas(&g_shader_program, george_texture_id, 0,
SPRITESHEET_DIMENSIONS, SPRITESHEET_DIMENSIONS);

    SDL_GL_SwapWindow(display_window);
}
```

And behold:

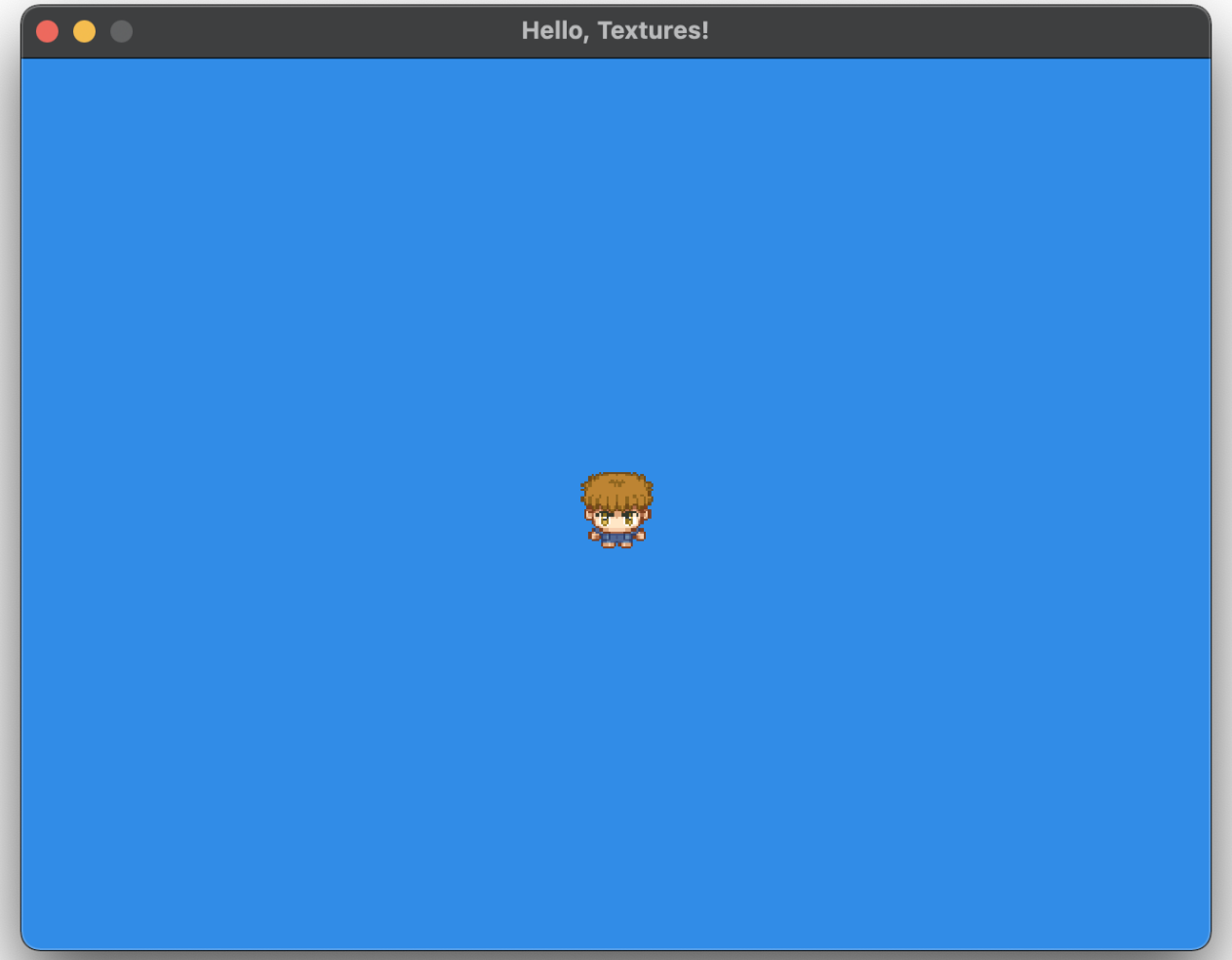


Figure 13: Only one section of our "texture" is being rendered.

Notice that I added the line `program.set_model_matrix(model_matrix)` to `render()` from `draw_object()`. If you don't do this, your sprite will not render as it will not have a model matrix to follow. The worst part about this error is that OpenGL doesn't crash because of it, and the warnings that it *does* give you are not specific at all. So don't forget!

Part 4: *Animation*

Now, this certainly is closer to what we want, but the whole point of this sprite sheet is for it to simulate a running motion by flipping through the different frames. This isn't terribly difficult; we simply need to tell OpenGL to choose a different index every X-number of frames:

```
constexpr int SPRITESHEET_DIMENSIONS = 4;
constexpr int FRAMES_PER_SECOND = 4;

int g_animation_indices[4] = {3, 7, 11, 15}; // for George to move to the left
int g_animation_frames = SPRITESHEET_DIMENSIONS;
int g_animation_index = 0;

float g_animation_time = 0.0f;


void update()
{
    /* Some code... */

    // Keep track of how much time has passed
    g_animation_time += delta_time;
    float seconds_per_frame = (float) 1 / FRAMES_PER_SECOND;

    // If we've reached the beginning of a frame span...
    if (g_animation_time >= seconds_per_frame)
    {
        // Reset the animation time, change the index
        g_animation_time = 0.0f;
        g_animation_index++;

        // And reset the index if we've reached the last animation index
        if (g_animation_index >= g_animation_frames)
        {
            g_animation_index = 0;
        }
    }
}
```

Code Block 5: See here that we are keeping track of the indices, the current animation index, and how much time has passed.

You can even make `FRAMES_PER_SECOND` to simulate running!

Of course, this only covers one of the four directions George has been prepared for. Let's take care of that:

```
constexpr int SPRITESHEET_DIMENSIONS = 4;
constexpr int SECONDS_PER_FRAME = 4;
constexpr int LEFT  = 0,
              RIGHT = 1,
              UP    = 2,
```

```
                DOWN  = 3;

constexpr int FONTBANK_SIZE = 16;

int g_george_walking[SPRITESHEET_DIMENSIONS][SPRITESHEET_DIMENSIONS] =
{
    { 3, 7, 11, 15 }, // for George to move to the right,
    { 1, 5, 9,  13 }, // for George to move to the left,
    { 2, 6, 10, 14 }, // for George to move upwards,
    { 0, 4, 8,  12 }  // for George to move downwards
};

int *g_animation_indices = g_george_walking[DOWN];  // start George looking
down

void process_input()
{
    // some code...

    const Uint8 *key_state = SDL_GetKeyboardState(NULL);

    if (key_state[SDL_SCANCODE_LEFT])
    {
        g_george_movement.x = -1.0f;
        g_animation_indices = g_george_walking[LEFT];
    }
    else if (key_state[SDL_SCANCODE_RIGHT])
    {
        g_george_movement.x = 1.0f;
        g_animation_indices = g_george_walking[RIGHT];
    }

    if (key_state[SDL_SCANCODE_UP])
    {
        g_george_movement.y = 1.0f;
        g_animation_indices = g_george_walking[UP];
    }
    else if (key_state[SDL_SCANCODE_DOWN])
    {
        g_george_movement.y = -1.0f;
        g_animation_indices = g_george_walking[DOWN];
    }

    if (glm::length(g_george_movement) > 1.0f)
    {
        g_george_movement = glm::normalize(g_george_movement);
    }
}
```

Code Block 6: Changing the walking animation direction depending on which button is being pressed. If the idea of creating a list of pointers (`GEORGE_WALKING_ANIMATIONS`) makes you nauseous, you don't have to use it. Manually changing the value of `animation_indices` in `process_input()` is totally fine.

Finally, we only want George to move when we're pressing down on the arrow keys. This is easily done by checking the length of the `movement` vector:

```
if (glm::length(g_player_movement) != 0)
{
    g_animation_time += delta_time;
    float frames_per_second = (float) 1 / SECONDS_PER_FRAME;

    if (g_animation_time >= frames_per_second)
    {
        g_animation_time = 0.0f;
        g_animation_index++;

        if (g_animation_index >= g_animation_frames)
        {
            g_animation_index = 0;
        }
    }
}
```

Code Block 6: As simple as this.

Part 5: *Text*

When picking out character sets, using those that are located and spaced in an organised way goes a long way when it comes to actually putting them on screen. Take a look at the sprite sheet in figure 9. The fact that the letters are placed in alphabetical order can help us make extracting these specific characters a matter of a simple loop—it also happens to follow ASCII values.

For text, it's actually pretty handy to use C++'s `std::vector` class to store our desired letters. Keep in mind that this class works much like a Python list and has nothing to do with the `gsl::vec` family of classes.

```
constexpr int FONTBANK_SIZE = 16;

void draw_text(ShaderProgram *program, GLuint font_texture_id, std::string
text,
               float font_size, float spacing, glm::vec3 position)
{
    // Scale the size of the fontbank in the UV-plane
    // We will use this for spacing and positioning
    float width = 1.0f / FONTBANK_SIZE;
    float height = 1.0f / FONTBANK_SIZE;

    // Instead of having a single pair of arrays, we'll have a series of pairs-
one for
    // each character. Don't forget to include <vector>!
    std::vector<float> vertices;
    std::vector<float> texture_coordinates;
```

```
    // For every character...
    for (int i = 0; i < text.size(); i++) {
        // 1. Get their index in the spritesheet, as well as their offset (i.e.
their
        //    position relative to the whole sentence)
        int spritesheet_index = (int) text[i];  // ascii value of character
        float offset = (font_size + spacing) * i;

        // 2. Using the spritesheet index, we can calculate our U- and V-
coordinates
        float u_coordinate = (float) (spritesheet_index % FONTBANK_SIZE) /
FONTBANK_SIZE;
        float v_coordinate = (float) (spritesheet_index / FONTBANK_SIZE) /
FONTBANK_SIZE;

        // 3. Inset the current pair in both vectors
        vertices.insert(vertices.end(), {
            offset + (-0.5f * font_size), 0.5f * font_size,
            offset + (-0.5f * font_size), -0.5f * font_size,
            offset + (0.5f * font_size), 0.5f * font_size,
            offset + (0.5f * font_size), -0.5f * font_size,
            offset + (0.5f * font_size), 0.5f * font_size,
            offset + (-0.5f * font_size), -0.5f * font_size,
        });

        texture_coordinates.insert(texture_coordinates.end(), {
            u_coordinate, v_coordinate,
            u_coordinate, v_coordinate + height,
            u_coordinate + width, v_coordinate,
            u_coordinate + width, v_coordinate + height,
            u_coordinate + width, v_coordinate,
            u_coordinate, v_coordinate + height,
        });
    }

    // 4. And render all of them using the pairs
    glm::mat4 model_matrix = glm::mat4(1.0f);
    model_matrix = glm::translate(model_matrix, position);

    program->set_model_matrix(model_matrix);
    glUseProgram(program->get_program_id());

    glVertexAttribPointer(program->get_position_attribute(), 2, GL_FLOAT,
false, 0,
                          vertices.data());
    glEnableVertexAttribArray(program->get_position_attribute());
    glVertexAttribPointer(program->get_tex_coordinate_attribute(), 2, GL_FLOAT,
false, 0,
                          texture_coordinates.data());
    glEnableVertexAttribArray(program->get_tex_coordinate_attribute());

    glBindTexture(GL_TEXTURE_2D, font_texture_id);
    glDrawArrays(GL_TRIANGLES, 0, (int) (text.size() * 6));

    glDisableVertexAttribArray(program->get_position_attribute());
```

```
    glDisableVertexAttribArray(program->get_tex_coordinate_attribute());
}
```

```
void render()
{
    glClear(GL_COLOR_BUFFER_BIT);

    g_shader_program.set_model_matrix(g_george_matrix);
    draw_sprite_from_texture_atlas(&g_shader_program, g_george_texture_id,
                                   g_animation_indices[g_animation_index],
                                   SPRITESHEET_DIMENSIONS,
SPRITESHEET_DIMENSIONS);

    draw_text(&g_shader_program, g_font_texture_id, "Hello, George!", 0.5f,
0.05f,
              glm::vec3(-3.5f, 2.0f, 0.0f));

    SDL_GL_SwapWindow(g_display_window);
}
```

Code Block 8: A trusty text renderer. I know that I provide these pre-made functions, but please make an effort to actually understand them!

Result:

Figure 13: *Hi*, says George.