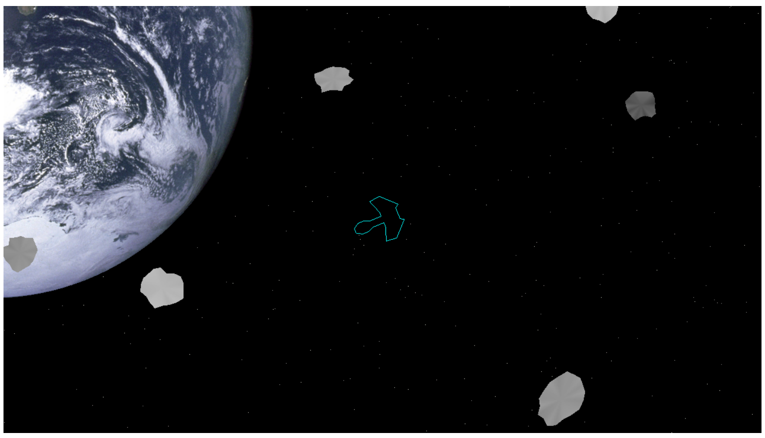
COMP3811 Coursework 1

Contents



Coursework 1 focuses on basic graphics operations in 2D, including manipulating images, drawing lines and triangles, and blitting images. Coursework 1 is to be solved individually and determines 30% of the total mark for COMP3811.

Before starting work on the CW, make sure to study this document in its entirety and plan your work. Pay particular attention to Section 1, which contains information about submission and marking.

1 Submission Instructions

Your submission will consist of source code and a report. Submissions are made through Gradescope (do *not* send your solutions by email!). You can use any of Gradescope's mechanisms for uploading the complete solution and report. In particular, Gradescope accepts `.zip` archives (you should see the contents of them when uploading to Gradescope). Do not use other archive formats (Gradescope must be able to unpack them!). See pages on Minerva for additional instructions. *The report is the basis for assessment. The source code is supporting evidence for assertions made in the report.*

The source code must compile and run as submitted on the standard SoC machines found in the UG teaching lab (2.05 in Bragg). Your code will be evaluated using automated tests, meaning that you must use the provided base code. You must not change the public interface that is defined in certain headers; this will be indicated via comments in the relevant files. Generally, you are not required to add additional sources or projects. Further, your code must compile cleanly, i.e., it should not produce any warnings. If there are warnings that you cannot resolve or believe are in error, you must list these in your report and provide an explanation of what the warning means and why it is acceptable in your case. *Do not change the warning level defined in the handed-out code. Disabling individual warnings through various means will still require documenting the warning in the report.*

The report must be a single PDF file called `report.pdf`. In this report, you must list all tasks that you have attempted and describe your solutions for each task. You may refer to your code in the descriptions, but descriptions that just say "see source code" are not sufficient. Include screenshots for each task (however, do *not* include screenshots/images of code - if you wish to include code, make sure it is rendered as text in the PDF using appropriate formatting and layout).

Apply good report writing practices. Structure your report appropriately. Use whole English sentences. Use appropriate grammar, punctuation and spelling. Provide figure captions to figures/screenshots, explaining what the figure/screenshot is showing and what the reader should pay attention to. Refer to figures from your main text. Cite external references appropriately.

Your submission must not include any "extra" files that are not required to build or run your submission (aside from the report). In particular, you must *not* include build artifacts (e.g. final binaries, `.o` files, ...), temporary files generated by your IDE or other tools (e.g. `.vs` directory and contents) or files used by version control (e.g. `.git` directory and related files). Note that some of these files may be hidden by default, but they are almost always visible when inspecting the archive with various tools. Do not submit unused code (e.g. created for testing). Submitting unnecessary files may result in a deduction of marks.

While you are encouraged to use version control software/source code management software (such as git or subversion), you must ***not*** *make your solutions publicly available. In particular, if you wish to use Github, you must use a private repository. You should be the only user with access to that repository.*

Note on plagiarism

You are allowed to discuss ideas with your colleagues. However do not share your code with anybody else. You must program independently and not base your submission on any code other than what has been provided with the coursework and/or in the exercises for COMP3811. As a special exception, you may reuse code from COMP3811 exercises that *you are the sole author of.*

You are encouraged to research solutions and use third-party resources. If you find such, you must provide a reference to them in your report (include information about the source and original author(s)). Never "copy-paste" code from elsewhere – all code must be written yourself. If the solution is based on third party code, make sure to indicate this in comments surrounding your implementation in your code, in addition to including a reference in your report. *It is expected that you fully understand all code that you hand in as part of your submission. You may be asked to explain any such code as part of the marking process. If deemed necessary, you may be asked to attend a short individual interview with the instructor(s), where you are asked to explain specific parts of your submission.*

Use good commenting practices to explain your approach and solution. Good, thoughtful and well-written comments will help you show that you understand your code. It will also decrease the chances of accidentally ending up with submissions similar to other's work.

Coursework 1 will not require you to use any third-part software outside of what is included in the handed-out code. You are therefore not allowed to use additional third-party libraries.

2 Tasks

Start by downloading the Coursework 1 base code. Make sure you are able to build it. If necessary, refer to the first exercise handed out in COMP3811. It uses the same base structure and includes detailed instructions to get you started.

Although the teaser image looks somewhat like a screenshot from a game, quite a few things that make a game are missing. This includes functionality like collision detection, sound, game logic, AI, networking, etc etc. However, most importantly for COMP3811, there are several graphics subroutines whose implementations are missing as well. Each task that you complete will progress you from the initial empty black screen towards the teaser image shown on the first page in this document.

Coursework 1 includes tasks for a maximum of 30 marks. Each of the tasks below indicates the maximum number of marks that you can receive for it. Grading of each task is assessed based on: code quality, including correctness, clarity, commenting and efficiency; and based on the descriptions and analysis in your report. Your code must work in both debug and release modes.

2.1 Setting Pixels

Drawing anything on screen ultimately requires you to set a specific pixel to a specific color. In this first task, you will implement helper functions to do so. Any drawing from here on out will use these helpers, specifically the `Surface:set_pixel_srgb` method.

Consider the `Surface::set_pixel_srgb` and `Surface::get_linear_index` methods. These are declared in the `Surface` class in `draw2d/surface.hpp` and defined in `draw2d/surface.inl`. Implement the two functions in `draw2d/surface.inl`.

The `Surface` class uses a RGBx image format, where each color component is stored in a single 8-bit unsigned

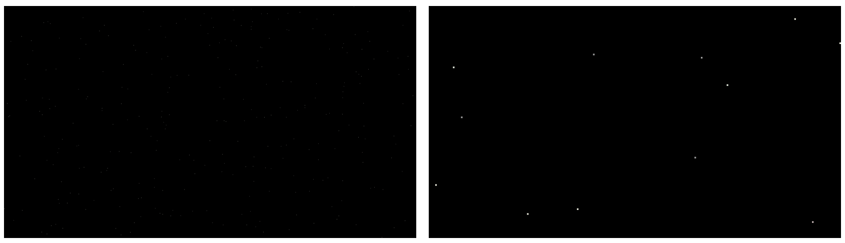

(a) *Background "starfield"* **(b)** *Magnified view*

Figure 1: *Task 1. You might need to zoom in to the left image in your PDF viewer to see the individual points. The right image shows a magnified view of the top-left region.*

integer (`std::uint8_t`). You may set the fourth component ("x") to zero. It is included to pad each pixel to be 32-bits but otherwise ignored. The image is stored in row-major order.

Important:

- You are *not* allowed to change the `draw2d/surface.hpp` header (and, consequently, you may not change the interface of the `Surface` class).
- You must keep the `assert()`-line as the first line of the `Surface::set_pixel_srgb` function definition. Add new code below it.

This methods are used to draw the background particle field. Refer to Figure 1 for possible results. You can move around by first tapping `space` to enter piloting mode (your mouse cursor should turn into a crosshair), moving the mouse cursor in the direction you wish to accelerate, and then pressing and holding the right mouse button to accelerate. Tapping space bar a second time will exit the piloting mode.

2 marks

2.2 Drawing Lines

Next, consider the function `draw_line_solid`. The function is declared in the `draw2d/draw.hpp` header and defined in the `draw2d/draw.cpp` source file. The function is supposed to draw a solid single-color line between the points `aBegin` and `aEnd`. The color of the line is specified by the function's final argument.

Implement the `draw_line_solid` function. The goal is to produce a line that is as thin as possible (single pixel width) and that does not have any holes (i.e., each pixel should connect to another pixel either by nearest neighbours or by diagonals). Recall the parametrised version of a line as a starting point. You should ensure that the function produces correct results with all inputs. Consider edge cases. For full marks, further consider efficiency. The line should be drawn in the minimum number of operations/pixel writes.

The handed-out code contains an additional program to test your line drawing, `tests-line`. It includes a small number of example cases. You can switch between the different cases using the number keys. See source code comments for brief descriptions of each test (`tests-line/main.cpp`). The included tests are just a few examples and are not exhaustive, so you may want to add your own cases.

In your report, explain your method and document what special cases you have found and can handle. Describe any additional test cases that you have identified.

Note: You must not change the prototype of the `draw_line_solid` function.

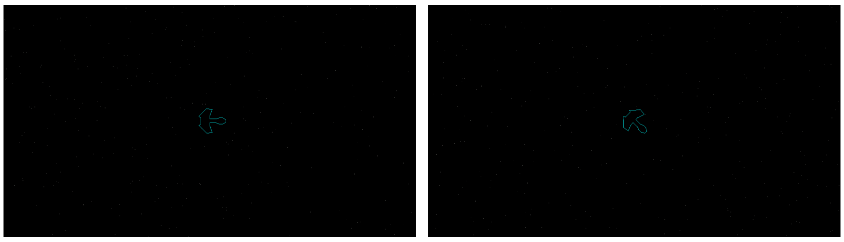
With the line drawing in place, you should now be able to see a space ship (Figure 2a).

6 marks

2.3 2D Rotation

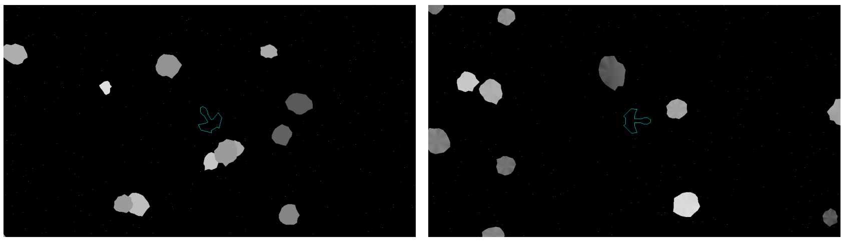
The space ship initially always faces to the right. To make it turn, you must implement a few functions related to the 2×2 matrices:

- Matrix-matrix multiplication: `Mat22f operator*( Mat22f const&, Mat22f const& ) noexcept`
- Matrix-vector multiplication: `Vec2f operator*( Mat22f const&, Vec2f const& ) noexcept`
- Creation of a rotation matrix: `Mat22f make_rotation_2d( float aAngleInRadians ) noexcept`



(a) *Section 2.2* **(b)** *Section 2.3*

Figure 2: *(a) Space ship without rotation, facing the default direction (right). (b) Space ship with rotation, always facing the mouse cursor when in piloting mode.*



(a) *Section 2.4* **(b)** *Section 2.5*

Figure 3: *(a) solid color asteroids. (b) asteroids with slight color variations (see, for example, the asteroid just to the top-left of the ship in the right hand side image).*

The functions are both declared and defined in `vmlib/mat22.hpp`. Provide implementations for these functions/operators.

With the implementations in place, the ship should now always face the mouse cursor when in piloting mode (Figure 2b).

3 marks

2.4 Drawing triangles with the half-plane test

Consider the function `draw_triangle_solid`. It is declared in the `draw2d/draw.hpp` header and defined in `draw2d/draw.cpp`. This function draws a single triangle defined by its three vertices (`aP0`, `aP1` and `aP2`), filled with a solid color. Use the half-plane test to determine if a point/pixel is inside the triangle.

Similar to Section 2.2, you should use the `tests-triangle` to test your triangle drawing in isolation. Make sure your method handles at least the included examples successfully.

In your report, explain your method. Document what special cases you have found and can handle. Describe any additional test cases that you have identified. Discuss the (theoretical) efficiency of your implementation and list any optimizations that you have performed.

Note: You must not change the prototype of the `draw_triangle_solid` function.

6 marks

2.5 Drawing triangles with barycentric coordinates

Consider the function `draw_triangle_interp`. Like the previous drawing functions, it is declared in the `draw2d/draw.hpp` header and defined in `draw2d/draw.cpp`. This function draws a single triangle defined by its three vertices (`aP0`, `aP1` and `aP2`). Each vertex is assigned a color (`aC0`, `aC1` and `aC2`, respectively). These colors should be interpolated across the triangle. Implement the function by computing barycentric coordinates for points/pixels. Use the barycentric coordinates to determine if a point/pixel is inside the triangle. If so, use the barycentric coordinates of the point to compute an interpolated color for that point.

Unlike earlier examples, the colors are specified in linear RGB (`ColorF`). You should perform the interpolation in linear RGB space and only convert to the 8-bit sRGB representation when writing the color value to the surface.

Run the separate tests as in Section 2.4. In your report, briefly describe your implementation and highlight the most important differences to the previous approach.

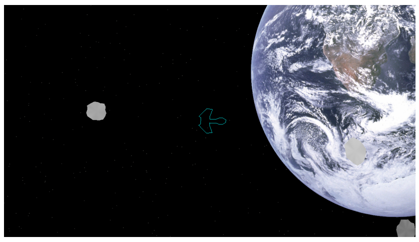


Figure 4: *Approaching the earth (lithobraking not yet implemented!).*

Note: You must not change the prototype of the `draw_triangle_interp` function.

6 marks

2.6 Blitting images

In the final task, you will implement image blitting with alpha masking. Consider the `blit_masked` function declared in `draw2d/image.hpp` and defined in `draw2d/image.cpp`. You will also need to implement a few helper functions in `draw2d/image.inl`. Search for lines containing the string *// TODO*.

You should blit the input image (`aImage` of type `ImageRGBA`) to the position specified by `aPosition`. Input pixels with an alpha value (`a` component of the `Color_sRGB_Alpha` color struct) less than 128 should be discarded.

If you have implemented the method correctly, you should find the earth after flying a bit to the right – it will be off-screen initially (see teaser image and Figure 4).

In your report, describe your implementation of the blit. Discuss the efficiency of your implementation. Are there any optimizations that you can think of?

Note: You must not change the prototype of the `blit_masked` function. You must not change the `ImageRGBA` class and the `load_image` function.

6 marks

2.7 Your own space ship

The default space ship shape is defined in `main/spaceship.cpp`. It consists of a number of points that are connected by lines.

Define your own custom space ship (see instructions in the source code). You must not use more than 32 points. The ship shape must show some amount of complexity and creativity. In your report, indicate if you have created a custom design and include a screenshot of your custom ship.

Please indicate in the source code (see comments) whether you would allow us to use your ship shape in future iterations of the COMP3811 module (for example as non-player ships). Your choice here does *not* affect the marking of this task.

1 mark

Wrapping up

Please double-check the requirements in Section 1 and ensure that your submission conforms to these. In particular, pay attention to file types (archive format and report format) and ensure that you have not included any unnecessary files in the submission. Make sure that you have tested your code (compile and run) in both debug and release modes.