Assembly Project: Dr Mario

Stefan Barna & James Han

Wednesday 26th March, 2025

1 Instruction and Summary

1. Which milestones were implemented?

   Milestones 1 & 2 have been implemented as of 2025/03/26. The collision detection and generation of new capsules has been completed for Milestone 3, however there is no clear or cascade support in the scenario where 4 or more same-coloured entities are aligned.

2. How to view the game:

   (a) Set the unit width and height to 1 px.

   (b) Set the display width and height to 256 px and 244 px, respectively.

   (c) If you are on a UNIX machine, change the relative file paths specified in the `Bitmap Assets` segment of the `.data` region to absolute file paths (from the root, not from the home directory).

3. Game Summary:

   What is being stored in memory? Our game uses memory to store repeatedly-accessed data, including loaded bitmaps, the current state of the game, and time-tracing values. Below is a description of each use.

   - Our playing field is represented by a 8×16 grid of tiles, each of which may be occupied by an entity, where we define an entity as a capsule half or a virus. To represent the state of the game, then, we must store the entities, and their positions, on the grid. We do this through a 128-byte array `BOTTLE`, so that each tile corresponds to precisely one byte. This way, we may query an entity at a given position in $\mathcal{O}(1)$.

     There are three primary attributes we must keep track of in an entity: the type (either a virus or a capsule half), the colour, and the direction the entity is facing. The final property is relevant only for capsule halves, and is used in rendering the entity and in determining the position of the paired capsule half. A capsule half without a pair is rendered differently and omits certain checks in cascade algorithms.

     To store all attributes in 1 byte, we partition the bits. The leading 4 bits represent the direction of the entity, and are ignored if the entity is of type virus. The next three bits correspond to the colour of the entity, which is one-hot encoded. The final bit is a toggle representing whether the entity is a virus (0), or a capsule (1). Any reference to an "entity byte" in this document refers to this single byte storing all properties of the entity. Working with entities then simply involves accessing the appropriate index of `BOTTLE` and using bitwise operations to extract or mutate desired properties.

   - We do not keep track of the player-controlled capsule in the `BOTTLE` grid, as this would involve moving the corresponding entity bytes to new locations upon movement, and resetting the previously-occupied entity bytes. Even then, we would need to keep track of the positions of the capsule externally. Instead, we store the positions (4 bytes each) and entity bytes (1 byte each) of each capsule half in memory, and update the position upon movement. When the capsule collides with the floor or another entity underneath it, these entity bytes are migrated to the `BOTTLE` at the stored positions.

   - In order to process higher resolution sprites efficiently, we use external software to draw the schematic, convert it to a bitmap (.bmp) file, then extract the pixel array from the bitmap into memory. Hence, in memory we must store the names of the bitmap files to be rendered as assets, used for `fopen` syscalls, and arrays to be populated by `read` syscalls for each asset. These are found in the `Bitmap Assets` segment of the `.data` region of our source. Please see Figure 1 as a demo of loaded pixel data.

Address	Value +0	Value +4	Value +8	Value +c	Value +10	Value +14	Value +18	Value +1c
0x1008875c	0x0	0x0	0xff574852	0xff574852	0xff574852	0xff574852	0xff574852	0xff574852
0x1008877c	0x0	0xff574852	0xff8caba1	0xffd2c9a5	0xffd2c9a5	0xff8caba1	0xff8caba1	0xff574852
0x1008879c	0xff574852	0xff8caba1	0xffd2c9a5	0xff8caba1	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e
0x100887bc	0xff574852	0xff8caba1	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e
0x100887dc	0xff574852	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e
0x100887fc	0xff574852	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e
0x1008881c	0x0	0xff574852	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff4b726e	0xff574852
0x1008883c	0x0	0x0	0xff574852	0xff574852	0xff574852	0xff574852	0xff574852	0xff574852

Figure 1: Left-direction blue half-capsule bitmap loaded into memory.

- To separate gravity effects from frame rate, we keep track of several variables in memory that concern the system time (`TIMESTAMP`), and the amount of time since gravity application (`DELTA`). `DELTA` is updated on every iteration of the game loop by adding the time difference between the current system time, determined with a syscall, and the previous time, stored in `TIMESTAMP`. Once it exceeds a particular limit, gravity is applied to the player capsule, or every capsule marked to fall during a cascade.

Figure 2: Screenshot of static scene, displaying bitmap assets.

2 Attribution Table

Stefan Barna (1010257758)	James Han
Bitmap Rendering	Entity Clear
Rendering Buffer	Cascade (after Clear)
Player Movement	
Collision Detection	
Gravity	