

Checkpoint 1

Due: Thursday January 22nd, by 11:59pm

Weight: 6%

ADDITIONAL DUE DATES

What is Due	Due Date
Team-up Deadline	Monday January 12th, by 11:59pm
Concept Shortlist Deadline	Thursday January 15th, by 11:59pm
Initial Draft Deadline	Thursday January 22nd, by 11:59pm

CONCEPT SHORT LIST

A short list of three (or more) potential game concepts must be submitted to your professor via email by the date and time specified at the start of this document. Rank these in order of preference, from highest to lowest. For each, provide:

- a one-paragraph summary of the game concept. For well known games this will be a short description, but for lesser known games or student created games sufficient detail will be required.

Following this submission, your professor will discuss these with you (and your partner(s), if applicable). At this time, your professor will either clear you to continue work on one of these concepts (perhaps with some requested changes or suggestions), or will require you to choose a new concept.

GAME SPECIFICATION

Overview

In this course you will develop a 2-D interactive video game of your choice. The first stage is to write a game specification, including all the details requested below. No programming is involved. The remaining stages (released throughout the semester) will deal with the game code design and implementation.

It is critical that the game be well thought out in advance, and that your specification be clear, precise and complete. Otherwise, your tasks on the remaining stages will be unclear.

You may invent your own unique game, or derive inspiration from a classic 2-D interactive video game (e.g. from the early 1980s or your mobile device). All submitted specifications this semester must be for different games. Pong will be used as an example throughout the course

so it is not eligible to be used. **Your professor must approve your game idea before beginning detailed work on your specification (see below).**

Keep your core game idea simple! Although not allowed, Pong is an example of a game of suitable complexity.

Group Work

Students may work in teams of three, or two, or individually. It is highly encouraged to work in a team – the project requirements remain the same whether you are in a team or an individual. Regardless of whether you decide to work in a team or individually, your *group* **must** inform your professor via email before the date and time specified at the start of this document.

Students who form a team **must** remain in that team for the duration of the semester, unless an explicit *divorce* is approved by the professor. If a team is divorced or if one partner withdraws from the course, the affected student(s) are responsible for completing all remaining stages individually.

The objective is that partners will put in equal effort. Grading of the project will be done in a demo and both partners are responsible for understanding all project code. The project gets one grade and failure to understand the project code by any partner will result in a grade deduction for the project!

Game and Specification Requirements

Games must satisfy the following general requirements. Additional details are discussed in the relevant sections which follow.

- Games must be based on animated, monochrome, 2-D graphics¹.
- Games must be interactive, with near-instant (*real time*) feedback to user input events.
- Both 1-player and 2-player versions must be specified.
- Games must include sound effects and music.
- Basic game play must involve user input from the keyboard. Optionally, mouse input may also be used for some user input during game play. Mouse input will be mandatory for a welcome *splash screen* in a later stage, but this can be ignored initially.
- The core game-play rules and graphics should be kept relatively simple! If extra elements are desired, it must be possible to add them only after the core game is working.

All of the following sections must be present in your specification document. All requested information must be provided. Supplementary sections and explanations may be included if judged necessary.

¹One of the best ways to compose 2-D graphics is to use *sprites*. Consult your professor if you are unsure what this means.

1 General Game Overview

A high level summary of the game concept. It must be possible to describe the essential concept clearly but concisely, in one paragraph. This must be accompanied by a hypothetical screen shot.

2 Game Play Details for Core 1-Player Version

This is the main section of the document. You must divide it into appropriate subsections for clarity.

2.1 Objectives and Rules

In detail, describe the game play. All rules (e.g. for scoring and winning, if applicable) must be clearly and explicitly stated. The game world physics² must be described, if applicable. This section must include at least one more hypothetical screen shot. Include additional diagrams if they help explain the game. A detailed description of how the keyboard and/or mouse is used for input is also required.

Note: There may be extra paragraphs here that correspond to the objects in the game where specific details on the objects, such as sizes and starting positions on the screen are included. How objects interact with the screen (i.e. movement rates) and with other objects (i.e. collisions) can be described here.

Citation Policy

Proper citations, in IEEE format, must be given for any images or other resources copied from elsewhere.

2.2 Game Objects and/or Object Types

Object types can be grouping of similar things (e.g. the player's ship, enemy missiles, etc.). When listing the objects in the game you will need to list the object's properties, the object's behaviours, and provide an image that shows what the object will look like. The properties can fall into two categories, the first category are the properties that describe the object physically, such as where it is on screen is on the screen (i.e. coordinates, size, etc), or any other physical attributes of the object (i.e. life, speed, direction, velocity etc.). The second category of property are those properties that describe how the first category are to change on the next render of the object. The behaviours are actions that the object can take (i.e. move, jump, shoot, duck) and not actions that happen to the object (i.e. die, fall, collide, etc.)

2.3 Game Events

List all game *events*. Each game event is to have a brief description. Three types of events must be considered:

²For example, how do balls bounce? How are collisions detected? Keep all game physics exceedingly simple! Pong is a good example of a game with appropriately simple physics.

1. Asynchronous, user input events (e.g. pressing the space bar to fire a shot);
2. Synchronous, timed events (e.g. every two seconds, an enemy ship appears);
3. Condition-based events (i.e. events that are triggered by the occurrence of other events, when some condition becomes true).

The game's response to, or behaviour at each event must be summarized.

IMPORTANT: The idea of objects and events in a video game is critical for design and implementation. Regardless of whether the game is programmed using an object-oriented language, which you will not be using, taking this perspective will make designing your project far easier.

2.4 Hypothetical Gaming Session

Explain the starting state of the game, and how typical play might progress. Note that some games are open-ended, while others have definite final goals.

3 Game Play Details for Core 2-Player Version

In detail, explain the 2-player version of the game and how it differs from the 1-player version. Any changes to the rules (e.g. for scoring and winning) must be stated. At least one hypothetical screen shot of the 2-player version must be included.

Note: in the 2-player version, each player will sit at their own computer. The two computers will be connected (e.g. via their serial ports). Play can be simultaneous, or can alternate between players; however, each player must be able to simultaneously see a shared world.

4 Sound Effects

It is hard to describe sound effects in writing. However, list a small number (e.g. two to four) sound effects which your game will incorporate. Describe the game events which will trigger each one. Note that sound effects must be kept basic (e.g. explosions, ball bounces, etc). It will not be possible to play simulated or recorded speech. At least one sound effect must be due to a synchronous event (or due to a condition-based event triggered by a synchronous event). At least one must be due to an asynchronous event (or due to an event triggered by an asynchronous event).

Your game must include background music. A simple, cyclic melody is sufficient. It is not necessary to describe the music in this document.

5 Additional Features

All of the above sections are for describing *core* game elements. Remember, the core game should be kept simple. It should either be absolutely minimal, or at least note the features which can be left out if you need to scale down the project.

In this section summarize any additional features which you would like to add if you have time. The core game should not rely on any of these being implemented.

Your specification must include sufficient detail, such that it should be possible to give your specification to a knowledgeable programmer and they should be able to create your game without making assumptions or ever having to talk to you.

Initial and Final Drafts

An initial draft of the game specification must be finished by approximately the date and time specified at the start of this document. This version is to be a complete draft of the game specification, with all sections and requested content provided.

After initial draft deadline, you (and your partner(s), if applicable) may be required to meet with your professor for a consultation, prior to completion of the final version. At this time, your professor may request changes or clarifications, and may make suggestions.

For this project, a precise word or page count is not being specified. However, it is difficult to imagine a complete document which is less than five pages long, including pictures.

Other Requirements

Specifications must be prepared electronically, in Word format (i.e. handwritten text, Google-Docs or any other electronic format will not be accepted). Hand-drawn diagrams can be used only if they are very neat and a scanned version is included in your document. You will be required to submit your specification via D2L.

The game specification will be graded as part of the final project marking process. Note that this means it is acceptable to make modifications to the specification throughout the semester, if necessary. The final submission will be graded based on:

- quality and appropriateness of game concept
- completeness of spec., including all required sections and details
- clarity and precision of spec.
- quality of English (spelling, grammar, etc.)
- quality of document structure, formatting and layout

Checkpoint 2

Due: February 23rd, by 11:59pm

OVERVIEW

In the previous checkpoint you developed a detailed specification for a simple 2D, monochrome video game for the Atari ST. **It is critical that the specification is as thorough, complete and clear as possible before proceeding.**

In this checkpoint you will develop the models that will be used by the game. After developing and thoroughly testing your models, you will design, implement, and test a raster graphics library to support the required graphics capabilities of your game. Lastly, you will implement a rendering library that will render the current state of the model to the screen using the raster library.

Testing

Making sure that all parts of the code in the game are adequately tested is important to ensure that when changes are made to code base that none of the existing functionality is broken when trying to add new functionality or fix a bug. On D2L there is a document on testing that must be used as a guide on what needs to be tested and how to build a test suite for a module. The test code for each module must test general cases and all edge cases to be considered a complete test module.

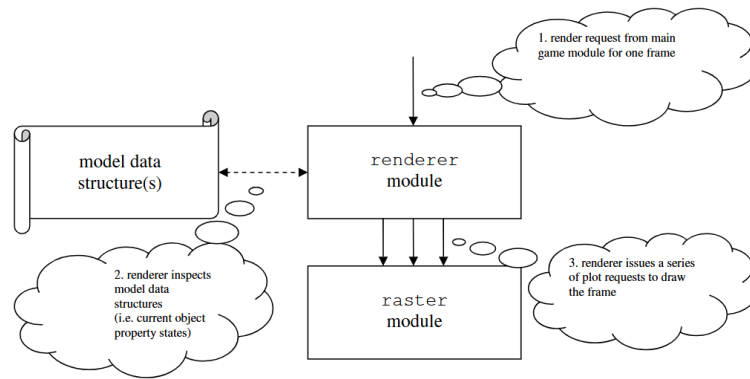
When writing tests, each module being tested will require its own corresponding test module. For the game, this means that for checkpoint 3, there will be three separate test modules, one corresponding to each game module being developed.

Background: The Design of Video Games, Part 1

A video game is based on a *model* which describes how the game world is represented internally by the computer. The model captures the essential properties of the world's objects, their behaviours and interactions, the game's logic and the world's physics. In other words, it describes how the world's state is represented at any point in time, as well as how the world's state evolves over time. ***The model is independent of how a view of the world is presented to the user*** (e.g. as an image on the screen). The model has very little or nothing to do with bitmaps, plotting algorithms or other aspects of how the objects are ultimately displayed.

Rendering is the process of generating an image from a model¹. In other words, the rendering module links the model module with the raster library. Later in this checkpoint will deal with rendering an individual frame of animation based on a static model state.

¹In 3D computer graphics, the term *rendering* has a more specific (but non-contradictory) meaning.



Model data structures are updated in response to events – perhaps asynchronous events triggered by key presses, synchronous events triggered by a clock tick, or conditional events triggered by either synchronous, asynchronous, or other conditional events. Checkpoint 3 will add event triggers and periodic render requests, so that the game world becomes animated².

THE MODEL

Overview

In this checkpoint the you will first design and implement the data structures for your game world's object types. At run time, objects will be represented by structures whose fields correspond to the objects' properties. Also, you will implement functions for manipulating objects according to their specified behaviours. Finally, you will implement event-handling functions for each of the specified asynchronous, synchronous and condition-based events.

Requirement: Model Data Structures

Object Type Models

Begin by designing the data structure(s) for representing the game world. You must identify each game world *object*³ and object type, and decide on a representation for each of its properties. Ideally, the important details have already been described in your Checkpoint 1 specification.

For example, in the specification a ball in Pong has position⁴ as well as velocity⁵ state. In C, this can be modelled as:

²Where the changes to the model are coming from are not shown on the above diagram as it is not relevant to the process of being able to render a complete frame.

³Although we are not using an object-oriented programming language, we will still think of the model as being comprised of various *objects*.

⁴Note: the Atari ST's monochrome monitor resolution is 640 × 400. The ball position is therefore within this area.

⁵Time-relative properties such as velocity should be measured against a 1/70th of a second clock rate, as discussed elsewhere.

```
typedef struct {
    unsigned int x, y;      /* type definition for Pong's ball object */
    int delta_x;           /* position coordinates */
    int delta_y;           /* horiz. displacement per clock tick */
} Ball;                   /* vert. displacement per clock tick */
```

Each object in your specification will be implemented in a corresponding module. For example, the pong ball would be implemented in a module `ball.h` with code in the file `ball.c`.

Use Integer Value Where Possible

Try to stick to integer values whenever possible. When representing fractions, it is often preferable to use a pair of integers instead of a floating-point value.

Avoid decimal numbers at all costs! The Atari 68000 chip does not have a built-in mechanism to handle floating-point numbers, which means for a game that wants to use decimals a floating-point library will need to be created.

Main Game Model

The main game will have a model associated with it, and it must have its definition in a file named `model.h` and any corresponding code in `model.c`.

If your game world has a large number of objects with heterogeneous types, collecting them by making intelligent use of arrays for multiple objects of homogeneous type. For example:

```
typedef struct {
    Ball ball;
    Paddle paddles[2];    /* 0=left, 1=right */
    /* ... etc ... */
} Model;
```

Later, it will be possible to declare instance(s) of the model type(s). Initialize them according to the desired initial state of the game world. For example:

```
Model testPongSnapshot = {
    { 320, 200, -2, 2 }, /* the single Ball instance */
    {
        { ... },         /* left paddle instance */
        { ... }          /* right paddle instance */
    },
    /* ... etc ... */
};
```

Requirement: Object Behaviour Functions

In the specification each object type has a set of behaviours. For each object add corresponding functions to the model for the object. For example:

```
#include "ball.h"

void move_ball(Ball *ball)
{
    ball->x += ball->delta_x;
    ball->y += ball->delta_y;

    /* ... what about collision detection? */
}
```

Also, declare (“prototype”) these functions in `ball.h`, because these will need to be callable by other modules.

Requirement: Event Handler Functions

The last part of creating the models is to develop an `events` module (using the provided `events.h`). This module will consist of three (3) submodules, one for each type of event. The three module will be `asynchronous` with corresponding files `asynch.h` and `asynch.c`, `synchronous` with corresponding files `synch.h` and `synch.c`, and `conditional` with corresponding files `cond.h` and `cond.c`. For each of these modules you will develop *event_handler* functions which update the model’s state for each event from the corresponding section of the game specification. The actual events won’t be occurring yet, but their handlers can still be implemented and tested. These functions will delegate some or all of their tasks to the relevant `model` functions.

Requirement: Test Driver

Write one or more test driver programs which help verify your model implementation, including event handling. Since you have not yet implemented a method of displaying the model, instead, using text output (e.g. the C `printf` function) is far better right now.

Each test program should place the object or objects in desired position(s) and then use calls to the various event functions to verify that any expected changes to the model occur. After the initial placement and after each action the value of *every* object property should be output. For example in Pong the paddle could be placed on the screen and then use the up and down arrow key event functions to indicate on the next *move* the paddle. Printing the paddle’s x and y positions will show whether the event function is working correctly. For synchronous events (i.e. the clock) use repeated calls to the synchronous event function, with output to indicate that a tick occurred, to indicate a tick. In Pong place the ball at a given location with a given velocity, loop to correspond to the number of ticks to be tested, each time through the loop call the keypress the ball should move. Printing the ball’s properties will show if the model’s properties are correctly updated. If the ball was placed close to a wall with the correct velocity it can be determined if it bounces correctly.

RASTER GRAPHICS LIBRARY

The raster graphics library is not responsible for animation. Its routines are for plotting static images like pixels, lines, shapes and/or bitmaps to the frame buffer. The graphics will become

animated in checkpoint 3.

Requirement: Provision of Low-Level Plotting Routines

The raster library is meant to be independent of your game, so you must to develop all of the following low-level routines, and perhaps others:

- `clear_screen`
- `clear_region`
- `plot_pixel`
- `plot_horizontal` line
- `plot_vertical` line
- `plot_line` (plot generic line between any two points)
- `plot_rectangle`
- `plot_square`
- `plot_triangle`
- `plot_bitmap_8`
- `plot_bitmap_16`
- `plot_bitmap_32`
- `plot_character`
- `plot_string`

These routines are “low level” from a design perspective because they will play a support role in rendering the game model to the screen, and elsewhere (i.e. other modules will invoke them to request fundamental screen drawing operations). They are also “low level” in the sense that they must not in turn rely on support from existing software modules. No operating system support is allowed in the implementation of your library, nor may it call third-party or standard C library functions. All plotting must be performed by your code, directly to the frame buffer.

Each function must take a *base* pointer as input, as well as function-specific input. The base pointer is the start address of the frame buffer into which the plotting must be performed.

Remember that your drawing routines must cope with screen boundary conditions. Forgetting about screen edges is a major cause of bugs.

For Full Marks

For all routines other than clear screen and plot pixel, if the image being drawn is requested to be drawn partially off the screen, then the routines must handle clipping, and if necessary for your game, wrap-around drawing of the raster.

Requirement: Game-Independence

Your raster graphics library must be independent of your game, in the sense that the library's design will not be coupled to game rules, object types, physics, events, etc.

For example, if the game was Pong, the raster graphics library would not *know* about paddles or balls, nor would it know about their properties or behaviours, ball movement or rebound physics, scoring rules, etc. If Pong's paddles and ball were each 4 bits wide, however, the library could include a *generic* routine for plotting *arbitrary* 4-pixel-wide bitmaps. This routine could later be called variously by other code to accomplish the plotting of paddles, the ball, and other 4-pixel wide images used in the game.

In theory, it should be possible to reuse the library for other games.

A Note On Text

Games often require the printing of text. This is easily accomplished using “bitmap” fonts: a bitmap (called a “glyph”) is provided for each character that might be plotted. Therefore, as long as bitmaps can be plotted, it will later be possible to print text in a straightforward way.

It is possible to design your own bitmap fonts (or maybe to find one that is free to use). Alternatively, it is possible to use the Atari ST's own bitmap font table. Your instructor will discuss how to accomplish this later. As well, I:\Labs\CompSci\Resources\2659\project\fonts folder contains a number of 8×16 font tables that you can use. For future reference, the font table is an array of 256 8×16 glyphs, indexed by ASCII value. If you plan to use these, then consider including a plotting routine for 8-bit-wide bitmaps as part of implementing the raster library for this checkpoint.

A Note on Smart Graphical Layout for Convenience and Efficiency

As was described in an earlier document, it is easiest (for architectural reasons) to base the game's graphics as much as possible on bitmaps which have widths of 8, 16 or 32 pixels (or multiples thereof) – ideally a suitable set of bitmaps have already been developed as part of your Checkpoint 1 game specification. If possible, it is convenient to arrange the frame buffer area so that the plotting of horizontal lines, bitmaps, etc. is aligned as much as possible on byte or word boundaries. When possible, this can lead to faster plotting and sometimes to simpler plotting algorithms.

The above are not possible in every game. For example, if some objects move smoothly and horizontally across the screen, then plotting (at least of those objects) will not always be byte-aligned.

Raster Test Driver

Your raster graphics library must be accompanied by a *test driver* program which invokes and thoroughly exercises each of its plotting routines.

Add tests for each function you write as you go. If you avoid prompt testing, it will be impossible to build robust code. Your tests should include testing a variety of scenarios and all boundary conditions.

A Note On Efficiency

At first, focus on the correctness of the plotting routines. Implement them in C, and avoid tricky optimizations.

Fast and Efficient Raster Algorithms

To decrease drawing speed, which will decrease rendering speed, and improve game play, all raster code should replace all multiplication, division, and modulus operations with other operations that require less processing time.

Once a routine is working correctly, it may then be possible to speed it up (sometimes by a large factor). Having a thorough set of tests will allow you to verify changes quickly and repeatedly.

Once a routine is working as quickly as possible in C, rewrite the routine in assembly language.

Note: It is not required that the plotting routines implemented in C before coding them in assembly, but that might make it easier.

For Partial Marks

An implementation of the raster library that is not completed written in assembly will not be eligible for full marks.

Skeleton Code

Your graphics library must be implemented as a C module named *raster* consisting of the following three files:

- `raster.h` ← header file describing the module's public interface, as needed by clients
- `raster.c` ← contains definitions for routines implemented in C (i.e. most of them)

- `rast_asm.s` ← contains definitions for routines implemented in assembly (if any)

To assist you, your instructor has placed a skeleton project in:

```
I:\Labs\CompSci\Resources\2659\project\stage2\skeleton
```

Also included in this folder are a make file for the project and a test driver skeleton.

A Note on the Proper Use of Header Files

A header file is intended to contain declarations relevant to a corresponding project module. Therefore, header files should contain information needed by client modules, including public function prototypes and public type definitions. Header files must *not* contain variable declarations or function definitions which result in space being reserved or in object code generation! Among other problems, this can lead to link errors (what if the same `.h` file is included in more than one `.c` module?). For some reason, a common student mistake is to define arrays for bitmaps in header files. Avoid this! Types, constants, functions, etc. which are purely for internal use in the module should similarly not be mentioned in the header.

RENDERER

Overview

In this stage, you will write a module which links the model with the low-level graphics library, so that it is possible to render an individual, static frame of animation based on a static snapshot of the world.

Before starting on this stage it is critical that the model's data structures are complete, well designed and implemented, and also that the raster module contains a reliable plotting routine for each type of graphics primitive (e.g. bitmap, shape, line, etc.) required by your game.

Once this checkpoint is complete, Checkpoint 3 will involve creating a main game module which updates the model based on user input and clock tick events, and that repeatedly renders frames of animation to achieve an animated, evolving game world.

Background: The Design of Video Games, Part 1 Recap

Recall that *rendering* is the process of generating an image from a model. Note that the main game module (which will be written in the next checkpoint) will eventually invoke the renderer module once for each frame of animation. In this stage, we are concerned only with rendering a single frame (i.e. a static, full-screen image) based on a static model (i.e. a snapshot of the model data structure(s) at a single point in time).

Requirement: Renderer Module

Provide a *render_object* function for each type of object which can appear on-screen. For example, Pong would include *render_ball* and *render_paddle* functions.

In addition, provide a master *render* function which renders a whole frame based on the current state of the model. Of course, it delegates to the various *render_object* sub-functions.

Each of these functions must take a frame buffer *base* pointer as input, as well as function-specific input. The base pointer points to the buffer into which the plotting must be performed. The function-specific input is the relevant portion of the model data structure(s).

For example:

```
void render(const Model *model, UINT8 *base);
void render_ball(const *Ball, UINT8 *base);
/* ... etc ... */
```

For Full Marks

For all object specific render functions (e.g. *render_ball* in Pong) must have as the parameter the object being rendered and **not** the overall game model. The master render function is the only render function that can have the game model as a parameter.

Requirement: Test Driver

The test driver for the rendering module is pretty simple. ***Make a copy of your model test driver***, and replace the printed output with calls to the master render function. If your model test driver is complete this will allow you to test the rendering code with a full set of different static model configurations. After each call to the master render function add code so that the tests wait for a keypress, and then clear the screen.

Minimizing Screen Plotting

In the first version of your renderer, it is acceptable to plot an entire frame each time the master render routine is invoked. In Checkpoint 3, an easy way to animate the game is to clear the screen and then completely re-render the entire game model, for each frame. Of course, the disadvantage of this approach is that it is slow. It is therefore difficult to achieve, smooth, high-quality graphics with a high frame rate.

Once basic rendering is working, you may wish to consider improving the renderer so that it only re-plots the aspects of the model that have changed since a previous invocation. There is more than one way to accomplish this – working out the details is your responsibility.

Minimizing screen plotting is not mandatory before starting Checkpoint 3; however, it is strongly recommended that you start thinking about how to do it. By the end of Checkpoint 3,

it is likely that your game's graphics will suffer very noticeably if it naively and inefficiently over-plots.

Background: Plotting Text using TOS's Bitmapped Font Table

For the plotting of text, it is possible to define your own “glyph” bitmaps – one for each character that may need to be plotted. These bitmaps can then be collected into an array of glyphs indexed by ASCII value. Assuming appropriate bitmap plotting routines exist, plotting strings of text is straightforward.

Alternatively, the Atari ST's operating system, TOS, has a bitmapped font table that you can use if you like. The font table is made up of 256 8×16 bitmaps – one bitmap per character in the extended ASCII character set. The following code shows how to access the table:

```
#include <linea.h>
#include <osbind.h>

...

UINT8 *base = Physbase();
UINT8 *fontTable;
char ch = 'a'; /* example character to plot */

linea0(); /* required TOS system call */
fontTable = (UINT8 *)V_FNT_AD; /* start address of font table */
```

Treat `font` as an array of bytes. Indexing it at the ASCII value of `ch` will give a byte. This byte is the first row of `ch`'s bitmap! *The second row is 256 bytes further down the table, and so on...*

```
*base = *(fontTable + ch);
*(base + 80) = *(fontTable + ch + 256);
*(base + 160) = *(fontTable + ch + 512);
...
```

For example, the following code plots `a` in the upper-left of the screen:
Of course, this can be accomplished more compactly using a 16-iteration loop.

Depending on how you have coded your bitmap plotting routines, it is likely that you expect bitmap rows to be contiguous, not 256 bytes apart! If you would rather that glyph rows be contiguous, write a program that *rips* the TOS font table and produces a modified font table with the glyphs stored as desired. In fact, this will be mandatory eventually, since a later checkpoint will eliminate reliance on TOS altogether. If you are printing strings, it is actually more convenient to leave the font table in its original form.

Other Requirements

Your code must be highly readable and self-documenting, including proper indentation and spacing, descriptive variable and function names, etc. No one function should be longer than approximately 25-30 lines of code – decompose as necessary. Code which is unnecessarily complex or hard to read will be severely penalized.

For each function you develop, write a short header block comment which specifies:

1. Its purpose, from the caller's perspective (if not perfectly clear from the name);
2. The purpose of each input parameter (if not perfectly clear from the name);
3. The purpose of each output parameter and return value (if not perfectly clear from the name);
4. Any assumptions, limitations or known bugs.

At the top of each source file, write a short block comment which summarizes the common purpose of the data structures and functions in the file. This should also include a normal ID header, but should also include team member names and the author's name.

You should add inline comments if you need to explain an algorithm or clarify a particularly tricky block of code, but keep this to a minimum.

Checkpoint 3

Due: March 16th, by 11:59pm

Overview

Before starting on this checkpoint, it is assumed that you have a working renderer module which can render an *entire*, single frame of animation based on the state of the model's data structures. This checkpoint will have you implementing, the input module, the main game loop and all of the game logic for the 1-player game, and then once that is completed you will implement double-buffering

This Checkpoint Takes Time

This checkpoint takes a significant amount of time as **ALL** game logic will be implemented as part of this checkpoint. Completing this checkpoint well is vital as the rest of the checkpoints require that the game logic is completed to be able to do subsequent checkpoints.

Ideally, if you have completed *Minimizing Screen Plotting* portion of Checkpoint 2, the renderer can also render an *updated* frame of animation (i.e. without redrawing the entire frame), based on information about which elements of the game model have changed since a previous invocation of the renderer.

In this checkpoint you will create a main game module. The module will contain a main game loop which polls for and detects clock ticks and keyboard input. These will allow the model to evolve in response to synchronous and asynchronous events, respectively. The main game loop will also periodically invoke the renderer, so that a sequence of frames of animation are produced over time. In this stage your game world ceases to be static, and becomes dynamic. In other words, your game will start to become playable!

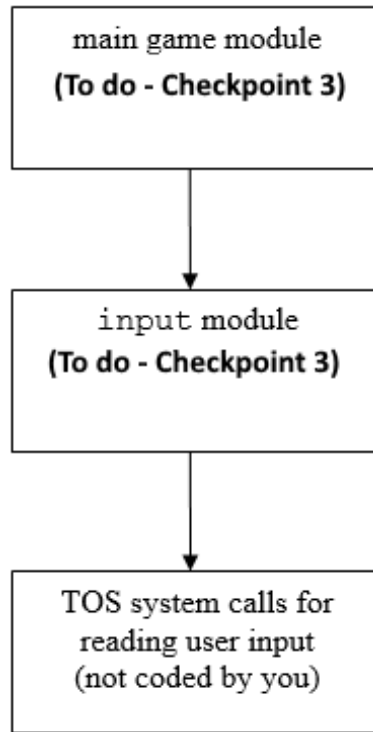
If you haven't yet completed *Minimizing Screen Plotting*, it is still possible to make progress on the main game module; however, the animation will be unnecessarily slow and choppy. Therefore, you will be forced to refine your renderer so that plotting is minimized. At the very latest, it will be necessary to address this issue prior to implementing double buffering.

Regardless of how well you complete the renderer, minimized screen plotting, and the main game loop, the animation will remain imperfect. You will probably notice graphical *flickering* and *tearing* effects. In short, this will be because plotting will not be synchronized with the screen refresh rate of the video hardware. You will fix this problem using a technique called *double buffering*.

INPUT MODULE

Background: Three-Level Input Design

When you code the user input portion of your game, you must follow a three-level design:



The middle layer contains the *high-level* input routines needed by the game. These in turn call the *low-level* input routines provided by the operating system. The former are hardware-independent, whereas the latter deal directly with the input hardware.

In summary, identify the input routines needed by your game and then implement a corresponding high-level input library named *input* with corresponding C files *input.h* and *input.c*. Note that these library routines must actually be game independent (i.e. it should be possible to reuse the library in other applications). These routines will wrap underlying, O/S-specific code for performing keyboard and mouse input. This will ensure that game code is as O/S-independent as possible.

The advantage of this approach will become apparent in later stages, when the O/S support will be replaced by custom low-level code. If the main game module code has not been decoupled from the O/S, it will make removal of the O/S support difficult. Instead, it should be possible to leave the game code alone and to only modify the middle layer wrappers.

Like for the previous stages, you must provide one or more test driver programs which invoke the middle layer routines.

Requirement: Input Module

You must create an input module. The input module will consist of two files `input.h` and `input.c`. The minimum functions required are a *has_input* function which returns true if there is pending input to be processed and false otherwise, and a *get_input* function which returns the next character to be processed.

Main Module & System Calls

Since the input module is the layer that communicates with the operating system for user input there must be **NO** system calls or functions wrapping system calls, related to user input, in the main game module!

ANIMATION AND GAME LOGIC

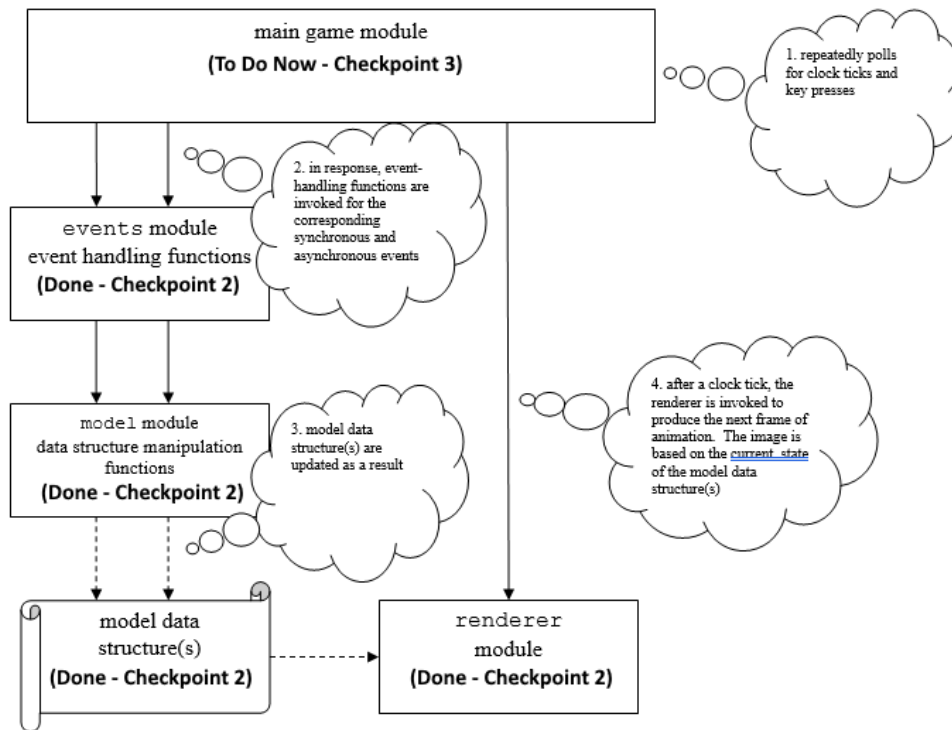
Background: The Design of Video Games, Part 2

In an animated video game, individual frames of animation are rendered in real time based on the current state of the model. The “frame rate” is the rate at which successive frames are displayed. The faster the frame rate, the smoother the animation will appear to be. For this game, frame rates up to 70 fps (frames per second) are achievable¹. This should appear smooth to most human observers. By contrast, frame rates down to about 12 fps may be perceived by most humans as animated, but jumpy².

A clock is used to drive the frame rate. It can also be used to produce synchronous (internally generated, clock synced) events within the game world. Of course, asynchronous (externally generated, non-clock synced) events such as user input must also be dealt with. Each event will typically entail updating some aspect of the model. Consequently, the next time a frame is rendered, the view of the world may have changed.

¹The Atari ST’s monochrome monitor has a 70 Hz refresh rate.

²Some commercial video games have frame rates in the 25-35 fps range. This is considered only minimally acceptable by some players. Other games attempt to match the refresh rate of the monitor, which is around 85 Hz on an average modern monitor. Still other games have an even higher frame rate, even though the monitor can’t match it. In this case, the extra frames are discarded (or the frames aren’t synchronized with the refresh rate, which leads to visual disturbances such as *tearing*).



An essential part of a game's code is its main game loop, found within the main game module. The algorithmic structure of this module is illustrated below:

```

initialize model
render model (first frame)
set quit = false

repeat until quit
  if input is pending
    process async event      <-- update model change requests
  if clock has ticked
    process sync events      <-- update model data
    process cond events      <-- update model data
  render model (next frame)

```

Note: Not every iteration of the loop will have an action being performed, as there may not be any unprocessed input nor may a clock tick have occurred.

Checkpoint Requirement

You are **REQUIRED** to use this main loop algorithm, without structural changes! It is critical to understand that asynchronous events update the model's request fields, i.e. pressing the up arrow key will set the direction to up. Pressing the up arrow **DOES NOT** move the item up. It is only on a clock tick that **ALL** game items are moved, i.e. data or position fields. Failure to follow this design will result in you needing to come back to this stage later in the semester and completely re-code from this stage onwards.

Requirement: Main Game Module

You must create a main game module. This must be coded as a source code file with an appropriate game-specific name (e.g. *pong.c*). This module must contain your game's main function. It may also contain various helper functions. Your makefile must generate a game-specific executable (e.g. *pong* or *pong.tos*).

The main game module must contain game initialization code, as well as the main game loop. The game is not to display welcome menus or splash screens at this stage.

TOS maintains a 70 Hz clock at memory location 0x462. Every 1/70th of a second, the longword value at this address is auto-incremented. The main game loop must repeatedly poll this value. Each time it changes synchronous events can be triggered. The rendering of the next frame of animation must also be triggered by clock ticks. You must not use user input events as rendering triggers!

To the main game loop, you must also add checks for user input so that asynchronous events can be triggered.

Background: How to Detect Clock Ticks

Important: The TOS maintains a 70 Hz timer: a longword at address 0x462 which is auto-incremented 70 times per second. Your game can monitor the passage of time by periodically reading the contents of this memory location.

The timer memory location is protected: the ST hardware monitors memory accesses at this address, and prevents them unless the CPU is in a "privileged" mode. Normally, only TOS code runs in this privileged mode, while ordinary user programs that attempt to access the timer crash with a bus error:

```
long *timer = (long *)0x462; /* address of longword auto-inc'd 70 x per s */
long timeNow;

...

timeNow = *timer;           /* crashes with bus error (2 bombs) here! */

...
```

However, TOS allows³ ordinary user programs to run in privileged mode! All that is required is a call to the `Super` system call:

```
#include <osbind.h>

...

long old_ssp;

old_ssp = Super(0);  /* enter privileged mode */
timeNow = *timer;
Super(old_ssp);      /* exit privileged mode as soon as possible */

...
```

The above code should be incorporated into a helper function called `get_time` which takes no parameters and returns the current timer value as a `UINT32`.

Aside: While it may seem like a good idea to run your entire program in privileged mode it is actually a terrible idea. When in privileged mode you have unfettered access to everything. Thus, if you inadvertently alter a system variable or peripheral the operation of the system can change, possibly causing the system to hang or crash. For example disabling the keyboard will result in the system appearing to hang. In non-privileged mode this action would have resulted in 8 bombs indicating a Privileged Violation. In privileged mode there will not be any bombs and you will have no idea what is the problem and where it occurred or why. Thus, it is common to change into privileged mode, perform only those actions that require privileged access and then leave privileged mode.

Once the above is working, it should be possible to incorporate code like the following into the main game loop:

```
UINT32 timeThen, timeNow, timeElapsed;

...

timeNow = get_time();
timeElapsed = timeNow - timeThen;

if (timeElapsed > 0) {
    ...          /* trigger synchronous events based on timeElapsed */
    ...          /* render model */
    timeThen = timeNow;
}
```

³Foolishly for TOS, but luckily for us.

Background: How to Detect Keyboard Input

TOS provides a `Cconis` system call for checking whether or not keyboard input is available for reading. And, it provides a `Cnecin` system call for actually reading it (without echoing the key press to the screen). Code like the following would read and process a key press only if one was pending:

```
#include <osbind.h>

...

if (Cconis()) {           /* is there user input read to be read? */
    char ch = (char)Cnecin(); /* if so, read it right away (otherwise don't */
    ...                /* waste time waiting for any) */
}
```

If `Cnecin` is called without first checking if a key has been pressed, it might *block* until a key is pressed.

More information on these TOS system calls is available in the reference subfolder of the course resources folder. The Devpac manual is a particular useful source of system call information.

Note: Some games require that key press and key release events be detected separately. Although this will be possible later, the routines above do not allow for this functionality, so these only key press events can be detected for now. However, TOS will cause a key that has been pressed and held down to auto-repeat. So, until a later stage, detection of auto-repeated key presses must temporarily be used in place of detection of a key release.

A Note on Detecting Mouse Input

The core game, as implemented in this checkpoint, is expected to be keyboard-driven. A welcome splash screen with a mouse-driven main menu will be added later. Therefore, mouse input detection is not normally required at this stage. Students who would like to incorporate mouse input now must consult with the instructor for more information.

DOUBLE BUFFERED GRAPHICS

Double buffered graphics will synchronize your game's frame rate (i.e. the rate at which successive frames of animation are produced) with the physical refresh rate of the video hardware and monitor. The goal is to achieve animated graphics that are free of visual distortions like flickering and tearing using a the *double buffering* technique.

Background: The Design of Video Games, Part 3

In computer animation, flickering and tearing occur when the frame rate is not synchronized with the refresh rate of the monitor. In other words, the video hardware is scanning the frame buffer at the same time as the frame buffer contents are being modified by the rendering routine.

To avoid this problem, it is necessary that these two processes occur at the same frequency (or that one frequency is a multiple of the other), and that they are in phase (i.e. that one doesn't start partway through the other).

One possible solution is to perform all the rendering for frames between screen refreshes, i.e. during the vertical blank intervals of a CRT monitor. This works well when the rendering process is very fast and can be guaranteed not to exceed the duration of the vertical blank (after which time the video hardware will begin scanning the frame buffer anew). However, this guarantee can be problematic in practice.

A better solution is to use the *double buffering* technique. A static image is held in the frame buffer while the next frame is rendered to a second "back buffer", off screen. Once this frame is complete, the *pages* are *flipped* at the next vertical blank. The former back buffer becomes the front buffer scanned by the video hardware during the next refresh, whereas the former front buffer can be used as the back buffer for rendering the next frame of animation.

It is important to note that the contents of the back buffer are *not copied* to the front buffer⁴. Instead, the video hardware register which contains the frame buffer start address is simply updated. The video hardware obtains the frame buffer start address from this video base register at each vertical blank, so that it is ready to start its next scan of the frame buffer at the correct location.

The use of double buffering does not guarantee smooth graphics. For example, if the rendering process was slow, at about 10 fps, the Atari ST's video hardware would perform about 7 screen refreshes for each page flip. Therefore, the animation would appear jumpy, albeit solid. In this scenario, effort would then need to be invested in optimizing the rendering process so that the animation was closer to (or ideally equal to) 70 fps.

⁴Copying an entire frame of memory can be as slow as rendering a frame, or slower if the rendering is optimized.

Requirement: Double Buffering

Double buffering must be implemented correctly, with absolutely no flickering or tearing. In this phase, all changes must be made to the main game module only.

First, it is necessary to allocate a 32,000 byte buffer. For hardware reasons that will be discussed later, frame buffers on the Atari ST must be 256-byte aligned (i.e. the start address must be a multiple of 256_{10})⁵. Initially, this buffer will serve as the back buffer (until the first flip).

Next, research the `Setscreen` system call (TOS XBIOS system call #5). This is the TOS routine which must be used to alter the frame buffer start address: it updates the video base register consulted by the video hardware at each vertical blank. Your main game loop must call `Setscreen` to schedule the flipping of the current front and back buffers each time the game program has finished rendering the next frame to the back buffer.

Common Problems

Stack smashing: the segment of memory reserved by TOS for the program's stack is not very large, and is less than 32,000 bytes wide. Therefore, the memory for the second frame buffer must *not* be allocated on the stack (i.e. as a local variable). This would cause the stack to overflow into the code segment. When the program began plotting to that frame buffer, it would corrupt existing code, resulting in odd behaviour and spurious crashes.

Incorrect parameters: In TOS, `Setscreen` can be used for more than one purpose. It takes three parameters, but only one of them (the physical frame buffer start address) is relevant to this project. The others should be passed as -1 to indicate that they are not being changed.

Premature rendering: `Setscreen` does *not* flip the pages immediately. When called, it updates the video base register so that the flip will happen at the next vertical blank, and then it returns immediately. Your code must therefore wait before starting to render the next frame. How long? Until after the next vertical blank has occurred⁶. If you forget this important detail and start rendering the next frame too soon, into what you think is the back buffer, you will still see flickering or tearing. This is because the buffer you *think* is the back buffer is really still the frame buffer, at least for a brief period of time.

Crash-on-quit: upon termination, the program must restore the original frame buffer. If not, 50% of the time, the program-allocated buffer will be at the front when TOS regains control. This region of memory is not where it expects the frame buffer to be, and strange behaviour and spurious crashes may result.

⁵Hint: allocate a region of memory slightly larger than 32,000 bytes, and then find an appropriate buffer start address within that range.

⁶Hint: it's easy to check this using the 70Hz *vertical blank counter*, i.e. the TOS-maintained longword at address 0x462 (**note:** this is the address used earlier). This counter is incremented at each vertical blank.

A Note on Achieving Smooth Animation at a High Frame Rate

The easiest way to implement double buffering is to erase the entire back buffer, and to fully render each frame, as described in Checkpoint 2 without completing the *Minimizing Screen Plotting* requirement; however, since this is the least time-efficient solution, the game will very likely not achieve a high enough frame rate. In other words, the animation will be flicker and tear-free, but may be *jumpy*. It is usually faster (although more complicated) to only re-plot the portions of the buffer which have changed. The problem when double buffering, though, is that the back buffer to be updated contains the view of the world from *two* frames ago.

One solution is as follows: right before rendering each frame, record a *snapshot* of the relevant parts of the model. The frame of animation in each buffer is paired with its corresponding snapshot, even though the model continues to evolve. When it comes time to render the next frame to the current back buffer, the current state of the world can be compared to the earlier snapshot to determine which portions of the buffer need to be re-plotted.

Other Requirements

Your code must be highly readable and self-documenting, including proper indentation and spacing, descriptive variable and function names, etc. No one function should be longer than approximately 25-30 lines of code – decompose as necessary. Code which is unnecessarily complex or hard to read will be severely penalized.

For each function you develop, write a short header block comment which specifies:

1. Its purpose, from the caller's perspective (if not perfectly clear from the name);
2. The purpose of each input parameter (if not perfectly clear from the name);
3. The purpose of each output parameter and return value (if not perfectly clear from the name);
4. Any assumptions, limitations or known bugs.

At the top of each source file, write a short block comment which summarizes the common purpose of the data structures and functions in the file. This should also include a normal ID header, but should also include team member names and the author's name.

You should add inline comments if you need to explain an algorithm or clarify a particularly tricky block of code, but keep this to a minimum.

Checkpoint 4

Due: March 23rd, by 11:59pm

OVERVIEW

Before starting this checkpoint it is assumed you have a functional main game loop and have implemented double buffering¹. In this checkpoint you will be implementing the sound effects, the background music, and the main title page².

MUSIC AND SOUND EFFECTS

The sound effect and music code should be completely tested and only once it is completely working should it be integrated into the working game, i.e. a working checkpoint 3. In other words you should code this stage separately from the rest of the project until such time as it is done, and then

Overview

In this stage you will add music and sounds effects to your game. First, you will implement a low-level PSG library. Then, you will use the PSG library to implement a *music* module. Finally, you will use the PSG library to implement an *effects* module.

An overview of the Atari ST's PSG is given in the lecture notes, and introductory exposure to programming the chip was given in a recent lab. It is your responsibility to review this material, and to consult the YM2149 documentation and an ST memory map for the important technical details.

Requirement: Provision of Low-Level PSG Routines

With the exception of the `Super` system call, you may not use any operating system support for the implementation of these routines, nor for the “music” and “effects” modules that are layered on top of it. All sound must be generated by programming the PSG chip directly.

You must first create a file module named `psg`. It must provide the following generic (game-independent) PSG control functions:

- `void write_psg(int reg, UINT8 val);`

Writes the given byte value (0-255) to the given PSG register (0-15). This is a helper routine to be used by the other functions in this module.

¹It is technically not required for double buffering to be implemented to complete this checkpoint, but not having implemented double buffering makes the implementation of this checkpoint harder.

²The main title page will only have keyboard input right in this checkpoint.

- `UINT8 read_psg(int reg);`

Useful for testing purposes, but unlikely to be used in either the music or effects modules.

- `void set_tone(int channel, int tuning);`

Loads the tone registers (coarse and fine) for the given channel (0=A, 1=B, 2=C) with the given **12-bit** tuning.

- `void set_volume(int channel, int volume);`

Loads the volume register for the given channel.

- `void enable_channel(int channel, int tone_on, int noise_on);`

Turns the given channel's tone/noise signals on/off (0=off, 1=on).

- `void stop_sound();`

Silences all PSG sound production.

Each function must validate its input parameters. If an input value is out-of-range, the function must ignore the request.

Requirement: Music

During game play a simple song must loop continuously³. Use PSG channel A for playing the melody⁴.

One simple way to implement a melody is as a circular array of notes (each with pitch data and maybe duration data). Use the 70 Hz vertical blank clock to time the playback of notes.

You must create a file module named `music` which provides the necessary functionality. The following design is suggested:

- `void start_music();`

Begins the playing of the song by loading the data for the first note into the PSG.

- `void update_music(UINT32 time_elapsed);`

Advances to the next note of the song **if necessary**, as determined by the amount of time elapsed since the previous call. The time elapsed is determined by the caller. It is intended that this value is equal to the current value of the vertical blank clock, minus its value when the function was last called.

³You can have different songs for different game stages if you want to get fancy. This is not required.

⁴If you wish to include a second musical line (e.g. a harmony), use channel B. This will limit sound effects to channel C.

If the above design approach is taken, the main game loop could be updated as illustrated below:

```
initialize model
render model (first frame)
start music
set quit = false

repeat until quit
    if input is pending
        process async event    ← update model
    if clock has ticked
        process sync events    ← update model based on time elapsed
        process cond events    ← update model based on time elapsed
        render model (next frame)
        update music          ← based on time elapsed
```

Requirement: Sound Effects

At least two significantly different sounds effects must be generated during game play. At least one sound effect must be due to a synchronous event (or due to a condition-based event triggered by a synchronous event). At least one must be due to an asynchronous event (or due to an event triggered by an asynchronous event). Use PSG channel C for playing sound effects⁵.

Sound effects **must** make use of the PSG's envelope generator in some way.

The following two functions must be added to the *psg* module:

- `void set_noise(int tuning);`

Loads the noise register with the given tuning.

- `void set_envelope(int shape, unsigned int sustain);`

Loads the PSG envelope control registers with the given envelope shape and 16-bit sustain.

You must create an `effects` file module. It must have one `play_effect` function for each type of sound effect in your game.

The event handling functions in the `events` module must be modified to play the relevant effect whenever its triggering event occurs.

⁵If you wish the playback of two sound effects to overlap in time, also use channel B. This will limit music to channel A.

Requirement: Sound Test Drivers

Your PSG module must be accompanied by a test driver program which invokes and thoroughly exercises its routines. Similarly, the *music* and *effects* modules must each be accompanied by a test driver.

Add tests for each function you write as you go. If you avoid prompt testing, it will be difficult to build robust code.

SPLASH SCREEN WITH MAIN MENU

Requirement: Splash Screen with Main Menu

You must provide a welcome splash screen which provides a main menu. At start-up, this allows the user to select between *1 player* and *quit* options at a minimum. Later, the splash screen will include a *2 player* option. When the game is over, the program may optionally return to this menu.

Display each menu option as a rectangular button. Later, it will be possible to make this menu mouse-driven. At this stage, the keyboard is to be used for menu selection.

The main menu must not be handled by the main game loop. Instead, the program's main function must call one function for handling the splash screen, and then a second function for handling the game itself. The latter function must contain the main game loop.

Time permitting, you may also wish to display a background image, high score list, etc. on this screen. But, these are not core features of the project.

OTHER REQUIREMENTS

Your code must be highly readable and self-documenting, including proper indentation and spacing, descriptive variable and function names, etc. No one function should be longer than approximately 25-30 lines of code – decompose as necessary. Code which is unnecessarily complex or hard to read will be severely penalized.

For each function you develop, write a short header block comment which specifies:

1. Its purpose, from the caller's perspective (if not perfectly clear from the name);
2. The purpose of each input parameter (if not perfectly clear from the name);
3. The purpose of each output parameter and return value (if not perfectly clear from the name);
4. Any assumptions, limitations or known bugs.

At the top of each source file, write a short block comment which summarizes the common purpose of the data structures and functions in the file. This should also include a normal ID

header, but should also include team member names and the author's name.

You should add inline comments if you need to explain an algorithm or clarify a particularly tricky block of code, but keep this to a minimum.

Checkpoint 5

Due: April 13th, by 11:59pm

OVERVIEW

You have now reached the point where you have a fully functional game. Before continuing, it is critical that you take this opportunity to review your work in all previous stages. Ensure that all requirements described in those stages are met. This includes ensuring each of the following:

- That all aspects of the graphics are complete, including flicker/tear-free double buffered animation
- That the rendering is efficient and the animation is smooth, at or approaching 70fps
- That all aspects of the music and sound effects are complete
- That the core 1-player game play and functionality are finished, from the user's perspective
- That the title screen which responds to keyboard inputs has been implemented
- That the game specification document from stage 1 is in-sync with the game itself (update the spec. and/or the game as necessary)
- That the code is well designed and clearly written, and conforms to all the code and documentation standards described in earlier checkpoints

This might look like the end of the line, but there are still improvements that can be made that will improve game play in the game. This checkpoint will focus on removing your game's reliance on the operating system by replacing almost all system calls in your code with your own custom code. This checkpoint will be broken into three different sections each of which will focus on eliminating one portion of your games reliance on the operating system.

At the completion of this checkpoint the only system call remaining must be the ***Super*** system call. The *Super* system call will be removed in one of the bonus checkpoints if you decide to attempt them. This implies that the code does not issue any system calls other than *Super*, nor does it rely on any operating system variables/tables. The TOS font table and key table are examples of the O/S variables/tables, and are therefore strictly disallowed (unless you've *ripped* your own copies).

This also implies that any library functions which cause TOS system calls to be invoked indirectly are also forbidden. In particular, you may not use I/O functions such as `printf` or `getchar`. You must also avoid `malloc` and `free`, since they may occasionally need to interact with the operating system. You must also avoid `time` (e.g. `time(NULL)`) as this relies on the

system time which is a TOS system call. The implementations of `srand`¹ and `rand` do not invoke the operating system, so their usage is safe. Similarly, string manipulation functions such as `strcpy` and `strcmp` are safe. If in doubt about a particular function, seek guidance from your professor before continuing to use it.

REPLACING VIDEO RELATED SYSTEM CALLS

Background: Atari ST Video Hardware Summary (Monochrome)

As discussed previously, the frame buffer (FB) is a 16,000 word region of RAM. It holds the bitmap for the video image to display on the monitor's screen. The ST's video hardware repeatedly indexes through FB, word-by-word. The SHIFTER chip serializes each successive word to produce a video intensity signal, which is output on a pin of the video port.

Actually, the SHIFTER does not index memory directly. Another chip, called the MMU (Memory Management Unit), is responsible for addressing RAM. Each indexed word is placed on the data bus by RAM, and is then latched by the SHIFTER.

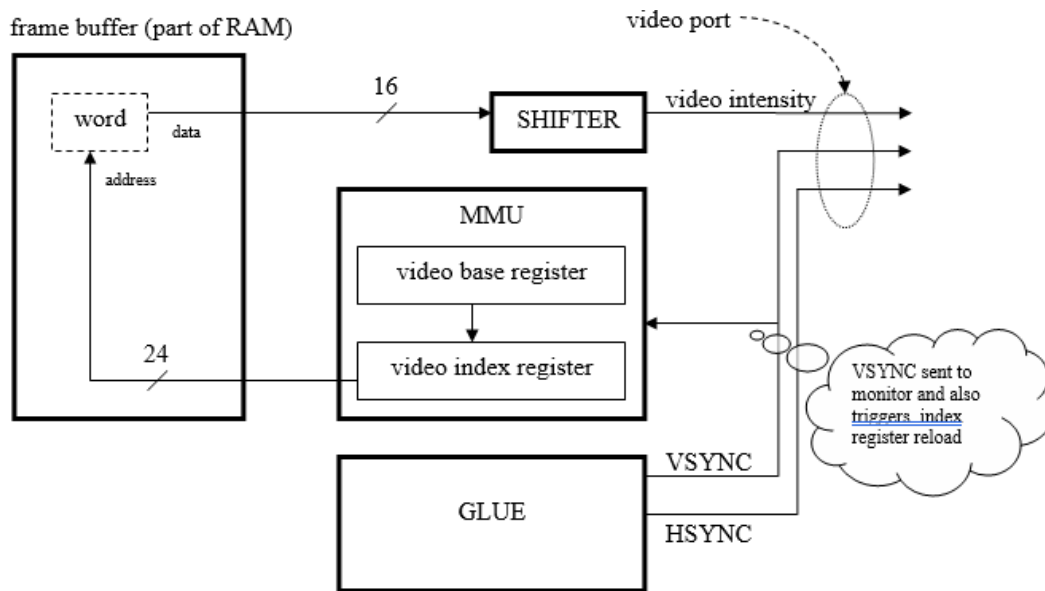
Yet another chip, the GLUE, is responsible for generating the horizontal and vertical synchronization signals (HSYNC and VSYNC) for the monitor. They specify the timing of horizontal and vertical scans. These two signals are output on the video port along with the video intensity signal. The entire screen is refreshed at a rate of 70 Hz.

Obviously, the MMU maintains a "video index register" to keep track of the current FB word to address. This 24-bit counter is periodically incremented by 2, so that it steps through the addresses of FB memory. The MMU also maintains a "video base register". This holds a pointer to the start address of the FB. At the end of each pass through the FB, the index register is reloaded with the contents of the base register in preparation for the next screen refresh.

Important: it is by altering the contents of the video base register that the frame buffer start address can be changed. This is how page flipping is accomplished when performing double buffered graphics. **Note carefully:** when a new value is written into the base register, the value is **not** loaded into the video index register until the next VSYNC! So, the page flip does not occur immediately. It occurs some time within the next 1/70th of a second.

¹As long as you are not doing an invocation of `srand` with `time` as the parameter.

The diagram below summarizes this arrangement:



One final comment for the curious: on the ST, the MMU handles all RAM accesses. If the 68000 places a valid RAM address on the address bus, the MMU forwards this address along to RAM. If the SHIFTER needs another word of data, again, the MMU addresses RAM at the next video index. The latter happens repeatedly, without the CPU's involvement. If the CPU and the SHIFTER both require memory to be accessed at roughly the same time, the MMU alternates memory access cycles between them.

Replace Physbase & Setscreen

In this section, you must remove the `Physbase` and `Setscreen` system calls, and replace them with your own implementations. You will create your own versions of the `Physbase` and `Setscreen` functions, called: `get_video_base()` and `set_video_base` respectively. These two functions will be placed in the *raster* module.

Replace Physbase

Write your own routine for retrieving the current frame buffer start address:

```
UINT16 *get_video_base();
```

Replace each call to `Physbase` with a call to `get_video_base`.

If you consult a memory map, you will discover that there is a 16-bit video base register (spread over two 8-bit addresses) which holds the frame buffer start address. These two bytes represent the high and middle bytes of the 24-bit video base. The low 8 bits are always zero, and are not stored².

Replace Setscreen

Once `get_video_base` is working, write your own routine for changing the frame buffer start address:

```
void set_video_base(UINT16 *);
```

Replace each call to `Setscreen` with a call to `set_video_base`.

Implementing these two functions in C does not require much code. However, to be fully correct, the latter function must really be implemented in 68000 assembly language using the `movep` instruction. This is so the two bytes of the 16-bit video base register are updated at the same time. Not doing so can trigger a bug if VSYNC happens midway through the `set_video_base` algorithm's execution, at the point when the two 8-bit portions of the video base register are inconsistent³.

CUSTOM ISRs

In this section you will replace two of TOS's ISRs with two of your own. Specifically, you will write custom VBL (vertical blank) and IKBD ISRs. We will refer to the development of each of these ISRs as two sub-sections. Budget about 1-2 days to complete the VLB ISR sub-section⁴, and 3-4 days for the IKBD ISR sub-section.

From the user's perspective, the impact of these changes on the game's functionality should be almost non-existent⁵. However, your program's reliance on the operating system will be eliminated almost completely!

General ISR Requirements (Applies to Both VBL and IKBD ISR Sub-Stages)

As you know, an ISR must be implemented in assembly language. However, it is acceptable to delegate one or more subtasks to subroutines written in C. Of course, this may not be necessary for very simple subtasks.

²This is why frame buffer addresses on the ST must be 256-byte aligned.

³This is from the fact that it would require two different `move` operations to change the video base register and an interrupt can be signalled before first has finished being executed but before the second, which means the VBL interrupt would occur with a partial address in the video base register

⁴If all previous stages have been designed and implemented well, the VBL ISR should actually be very quick.

⁵One exception is that the text cursor maintained by TOS will cease to blink after implementing the VBL ISR. This is because the plotting of the text cursor is performed in the TOS VBL ISR.

ISRs *must always* be as *fast* as possible! The idea is to do the minimum amount of work and then return quickly to the interrupted task. Remember that other, lower priority (but still important) ISRs may be delayed due to the execution of your ISR.

Time-consuming operations must be left for the main program to perform. Typically, an ISR will simply update some global data (if necessary) and then will somehow notify the main program (e.g. by setting a flag or enqueueing in a buffer) that some extra work needs to be performed.

Ideally, your game supports the ability to quit back to TOS. When an ISR is installed, the previous vector must be saved so that it can be restored on exit.

Requirement: VBL ISR

The goal of this sub-stage is to implement a VBL ISR for your game. This will enable you to reduce the amount of clock polling performed by the main game loop.

You must replace the TOS VBL ISR with your own. It must:

⁶

- Time the page flipping for double buffered graphics
- Time the playing of music
- Time any other synchronous/conditional events needed by your game
- Time the update of the game model based on any asynchronous events since the last vertical blank

Guidance on how to accomplish each of the above are given in the next three sections.

Vertical Blank Notes

Since your game will have a maximum frame rate of 70 Hz, there is no point in generating synchronous events using a faster timer. Put another way, why update your model more frequently than you can render views of the model? Regardless, some games may require a higher frequency, e.g. for measuring a user's reaction time. In this case, you can use one of the ST's programmable hardware timers. However, such issues will *not* be dealt with in the IKBD ISR sub-stage and will not be discussed here. Seek additional guidance from your instructor as necessary.

⁶Since your game will have a maximum frame rate of 70 Hz, there is no point in generating synchronous events using a faster timer. Put another way, why update your model more frequently than you can render views of the model? Regardless, some games may require a higher frequency, e.g. for measuring a user's reaction time. In this case, you can use one of the ST's programmable hardware timers. However, such issues will *not* be dealt with in the IKBD ISR sub-stage and will not be discussed here. Seek additional guidance from your instructor as necessary.

Your main game loop must **not** perform any clock polling. This includes no longer consulting the TOS vertical blank counter (the longword at address \$462, sometimes referred to as *vbclock*)⁷.

VBL ISR & vbclock

Your custom VBL ISR **MUST NOT** simply increment the vbclock. This will result in severe mark reduction for the VBL ISR portion of the grade.

Guidance: Timing of Page Flipping (VBL ISR)

One way of achieving flicker and tear free animation is to perform all screen drawing inside the VBL ISR, i.e. between screen refreshes, when the frame buffer isn't being scanned. However, this only works for very short plotting operations. This technique will not usually work for games such as yours because the image rendering takes too long. If your code is still updating the frame buffer when the next screen refresh begins, then image distortion will be seen.

Another common (but incorrect) approach is to render the next frame of animation in the main routine, but to then flip the buffers inside the ISR. The idea (again, incorrect) is that page flips will be synchronized with screen refreshes. This will not work because the video index register is reloaded from the video base register at the same time as the VBL IRQ is asserted. By the time your ISR is invoked, it is too late to update the video base.

The correct technique for your game is as follows: 70 times per second, the VBL ISR requests that the next frame of animation be rendered, e.g. by setting a global *render request* flag. When the main game loop sees that "render request" has been set, it calls the master render routine to render the next frame of animation into the back buffer. Then, a page flip is scheduled by calling `set_video_base`. The *render request* is then cleared. Note that the pages won't actually flip until the next vertical blank (when the video index register is reloaded with the new video base). If the main game loop is polling *render request* properly, it won't attempt to produce a subsequent frame of animation until after that flip has occurred.

In pseudo-code:

VBL ISR⁸:

```
update music (as necessary)
process synchronous/conditional events
update model based on asynchronous event requests (as necessary)
set render_request = 1
```

⁷In fact, vbclock is updated by the TOS VBL ISR. So, this clock will cease to tick once the operating system's own VBL ISR has been replaced.

⁸Note: the time elapsed since the last vertical blank, as measured in 70Hz clock ticks, will always be 1.

main game loop:

```
repeat until game over

    if asynchronous event(s) outstanding
    then
        process and update model as necessary

    if render_request
    then
        render model to back buffer
        schedule page flip by calling set_video_base
        set render request = 0
```

Guidance: Timing of Music/Sound Effects (VBL ISR)

Instead of polling the TOS vbclock in the main game loop to decide when it is time to move on to the next note, simply update the music within the VBL ISR. When it is time to move on to the next note, the ISR (or a delegate subroutine such as `update_music`) will simply update the state of the PSG. Since this is a relatively quick operation, it can be done exclusively within the ISR. There must be no song-related event processing left in the main game loop at all.

Be sure to initialize all song-related data structures and variables before installing the VBL ISR.

Note that the playing of sound effects will not necessarily be handled by the VBL ISR, unless they are in response to synchronous events. Sound effects which play in response to asynchronous events (e.g. user input) will be triggered elsewhere.

Important: each region of code which accesses the PSG is a critical section – figure out why⁹! Any main program code which attempts to access the PSG **must** do so with VBL interrupts masked.

Guidance: Timing of Other Synchronous Events (VBL ISR)

Likely, your game must process other types of synchronous events, e.g. enemy ships appearing or balls moving. Use the VBL ISR to trigger these events.

For example, if a ball is moving across the screen at constant velocity then update its position regularly, inside the VBL ISR. Do not couple the position updating with rendering. These are separate tasks. Note also that rendering time may not be constant; it may vary depending on the complexity of the image being produced. For the ball to move at a constant rate, its position must be updated regularly within the ISR, regardless of what the main program is busy doing.

⁹Hint: what could happen if the main program was in the middle of modifying PSG registers for a sound effect, due to an asynchronous user input event, and then the VBL ISR interjected and tried to play the next song tone? In particular, think about how the PSG does register selection.

IKBD ISR Sub-Stage

The goal of this sub-section is to implement an IKBD ISR for your game. It will take care of all keyboard and mouse input incoming from the IKBD's 6301 microcontroller.

Requirement: IKBD ISR (IKBD ISR)

You must replace the TOS IKBD ISR with your own, which. It must:

- handle all key make and break codes incoming from the IKBD;
- handle all mouse packets incoming from the IKBD.

Guidance on how to accomplish each of the above are given in the following sections.

Guidance: Basic Issues Related to Incoming IKBD Data (IKBD ISR)

Under TOS, the Atari ST uses vector #70 for IKBD IRQs. The IKBD ISR will be invoked once for each byte received by the IKBD 6850 ACIA, as sent by the IKBD's 6301 micro-controller. Note that it may also be invoked in other circumstances, such Rx overruns. A complete ISR must handle these eventualities.

Don't forget that the IKBD ISR will be invoked once for each key make code, once for each key break code, and three times for each mouse packet. Therefore, so that it doesn't confuse mouse data with scancode data, it must maintain a state. In other words, the ISR will be a little software finite state machine.

Important technical detail: the last step performed by the IKBD ISR must be to clear bit #6 of the "in service B" register of the MFP 68901 chip¹⁰, just before returning. Otherwise, the ISR will run a maximum of one time and will never be invoked again. The reason for this will be discussed further in lecture.

Remember that the ST has two 6850s: one is the IKBD interface and one is the MIDI port controller. They share a common vector #, and hence a common ISR. In addition, there is always the danger of spurious interrupts¹¹. Therefore, a complete ISR must check that the IKBD 6850 is actually asserting an IRQ, by checking its status register, before it attempts to service an interrupt.

For this assignment, the safest thing to do is probably to disable MIDI 6850 interrupts at the start of the program, and then to re-enable them at the end.

Important: in this assignment, you may write various routines whose execution may overlap in time (e.g. ISR code vs. interruptible non-ISR code). As discussed in class, ensure that

¹⁰This register is mapped at address \$FFFA11.

¹¹Spurious interrupts are due to hardware problems such as excessive electromagnetic interference: noise might induce a signal on an IRQ line even though no device is asserting IRQ. In a well designed system, of course, the danger of interference should have been minimized. At any rate, STEem is an emulator and probably doesn't simulate spurious interrupts. Still, your ISR should assume the possibility and deal with them if they arise.

“critical sections” are protected. This requires masking keyboard interrupts at the start of the critical section, and unmasking them at the end. To avoid overly restrictive masking, mask these interrupts at the 68901 MFP, not at the 68000.

Guidance: Handling Incoming Make and Break Codes (IKBD ISR)

Should the ISR perform input event handling for key presses? Only if the event handling logic (i.e. the required update to the game world model) is **very** simple and quick. This is discouraged, since overruns becomes very likely.

IKBD event handling should usually be left to non-ISR code. In this case, the ISR must enqueue each key press's scan code in a buffer (a circular queue implemented as an array). As discussed in class, the main game loop must repeatedly check to see if the buffer is non-empty, and if so it must dequeue the key and process the event.

Guidance: Managing Key Auto-Repeat (IKBD ISR if necessary)

In some games, key presses are used for “discrete” input events, meaning that it is the depressing of the key which is important, not how long it is held down before being released. For example, a single shot might be fired from a ship each time the ‘f’ key is struck. The key must be re-struck in order to fire the next shot.

In other games, when a key is depressed, it signals the start of a “continuous” input event which is effect until the key is released. For example, as long as the “up arrow” key is held down, a ship might ascend. When the key is released, the ship stops ascending.

If your game takes input from the keyboard, it must decide whether input is discrete or continuous, and it must handle key make and break codes appropriately. They are both fairly straightforward to implement.

For continuous input, note that a timer ISR such as the VBL ISR will be involved. The exact approach will be game-specific. One common technique: as long as the break code for the most recent make code has not been received, regenerate the corresponding user input event at each timer ISR invocation. In other words, changes in key state are asynchronous events, but while the key is depressed a series of synchronous events are generated.

Guidance: Handling Incoming Mouse Packets (IKBD ISR)

Your IKBD ISR must deal with mouse packet bytes, even if the mouse isn't used during game play. One reason is that mouse packet bytes should not be misinterpreted as key make or break codes if the mouse is jiggled during play. Another is that the splash screen in the final version of your game must be mouse-driven.

To process mouse input, maintain global mouse x and y coordinates. Each mouse packet will contain three separate items. The receipt of each by the 6850 will result in separate interrupts. Do **not** wait in the ISR for the arrival of all three bytes. It is guaranteed that the three items will be consecutive, i.e. a make or break from a key press will never occur between the items

of a mouse packet. The three parts of a mouse packet are the header and relative mouse movement data. The header indicates which if any mouse buttons are pressed and its value is 1111 10LR, where L means left mouse button and R means right mouse button. The relative mouse movement is the signed change in the x and y positions in pixels since the previous packet (these are commonly referred to as deltas: delta x and delta y). These deltas should be added to the global x and y coordinates. Be careful: the global coordinates will need to be integers, so your code will need to widen the deltas properly before adding.

IMPORTANT: drawing of the mouse is NOT to be done in the IKDB ISR. How this is handled will be covered in the next section.

COMPLETING THE MOUSE-DRIVEN SPLASH SCREEN

In an earlier stage you should already have completed a splash screen with a keyboard-driven main menu. You will now revise this menu so that it is mouse-driven.

Requirement: Splash Screen with Mouse-Driven Main Menu

You must provide a welcome splash screen which provides a main menu. At start-up, this allows the user to select between *1 player* and *quit* options at a minimum. Later, the splash screen will include a *2 player* option. When the game is over, the program may optionally return to this menu.

Display each menu option as a rectangular button. The user must be able to select an option by left-clicking within the corresponding rectangle.

When the splash screen is active, the program must maintain a mouse pointer plotted at an appropriate screen location, based on the user's mouse movement activity. The mouse pointer must not be visible during game play. When visible, the pointer must not flicker or tear.

The main menu **must not** be handled by the main game loop. Instead, the program's main function must call one function for handling the splash screen, and then a second function for handling the game itself. The latter function must contain the main game loop.

Time permitting, you may also wish to display a background image, high score list, etc. on this screen. But, these are not core features of the project.

Guidance: Mouse Input and Pointer Plotting

If the IKBD ISR from the previous section is fully and correctly implemented, the ISR should already be reading (and discarding) mouse packets. So that the program can keep track of the mouse's absolute screen location, maintain global x and y integer variables. Within the ISR add the Δx and Δy from each mouse packet to the global x and y variables. Similarly, a global variable can be used for maintaining mouse button state as obtained from each mouse button's header byte. The splash screen handling function, after plotting the menu, can continuously poll the button state. At each down-click, the mouse position can be consulted to determine

which menu option was selected (if any).

The main program **MUST NOT** access the mouse's absolute screen location via the global `x` and `y` variables. These should be accessed via functions, which you will create in the *input* module. This is identical to accessing the keyboard input, the main program does not access the global keyboard buffer, but rather uses appropriate system calls. Your mouse device driver, ISR and system calls, must be independent of your game program. You should be able to include only your device driver into any other program.

Two approaches may be considered for plotting a solid (non-flickering or tearing) mouse pointer. The first is to use double-buffering, which may be overkill. The second is to single-buffer the splash screen, and to very quickly plot the mouse pointer inside the VBL ISR¹².

When the mouse pointer has moved, it will be necessary to un-plot at the former location and to re-plot at the new location. One simple approach is to perform an XOR operation for both plotting and un-plotting. Alternatively, a “back fill” bitmap, of the same dimensions as the mouse pointer bitmap, can be maintained. It is used to hold a copy of the frame buffer region currently obscured by the pointer. When the pointer moves on, the uncovered region can be restored. Of course, a newly obscured region must be saved for later back filling.

Other Requirements

Your code must be highly readable and self-documenting, including proper indentation and spacing, descriptive variable and function names, etc. No one function should be longer than approximately 25-30 lines of code – decompose as necessary. Code which is unnecessarily complex or hard to read will be severely penalized.

For each function you develop, write a short header block comment which specifies:

1. Its purpose, from the caller's perspective (if not perfectly clear from the name);
2. The purpose of each input parameter (if not perfectly clear from the name);
3. The purpose of each output parameter and return value (if not perfectly clear from the name);
4. Any assumptions, limitations or known bugs.

At the top of each source file, write a short block comment which summarizes the common purpose of the data structures and functions in the file. This should also include a normal ID header, but should also include team member names and the author's name.

You should add inline comments if you need to explain an algorithm or clarify a particularly tricky block of code, but keep this to a minimum.

¹²Remember, the VBL ISR runs in response to a vertical blank – the brief period of time between screen refreshes. If the plotting operation is completed quickly enough, the entire mouse pointer bitmap will have been plotted before the next screen refresh begins.