

Atari ST Fruit Basket

A fully functional double-buffered 2D arcade game built in **C and 68000 Assembly** for real Atari ST hardware. The player controls a basket to catch falling fruit while dodging bombs. Difficulty scales dynamically as time progresses.

Gameplay

- Move the basket left (A) and right (D) to catch falling fruit
- Avoid bombs -- a single collision triggers the game over sequence
- Fruit and bomb spawn rates and velocities increase as time progresses
- Score increments on each successful catch and is displayed in real-time
- Splash screen on launch with animated text fade-in
- Game over screen shows final score with a descending melody

Technical Highlights

Double Buffering

The game maintains two framebuffers -- a front buffer displayed on screen and a back buffer rendered into off-screen. Every tick of the Atari ST's 70Hz TOS clock, the buffers are swapped via `Setscreen`, eliminating screen tearing entirely. The back buffer is 256-byte aligned via pointer arithmetic to meet Atari ST hardware requirements.

```
Setscreen(backBuffer, frontBuffer, -1); // swap front and back
```

After the swap, the back buffer is fully cleared before the next frame is drawn, keeping rendering clean without ghost artifacts.

Custom Raster Graphics Engine (RASTER.C)

All drawing is done via direct memory writes -- no OS graphics calls. The raster module implements:

- `plot_bitmap_16` -- 16-bit wide bitmap blitting for fruits and bombs
- `plot_bitmap_32` -- 32-bit wide bitmap blitting for the basket
- `plot_text` -- custom bitmap font renderer using a hand-coded font table (`font.c`)
- `clear_screen` -- full screen wipe via 32-bit word-level memory fill (8000 longwords)
- Direct pixel plotting by computing byte offset and bit mask from (row, col) coordinates

Dirty-Rect Optimized Renderer (`renderer.c`)

The renderer compares the current game state to the previous frame and only redraws objects whose position changed. This avoids redundant blitting and keeps frame updates efficient on constrained hardware.

```
if (game->basket.x != prev_game.basket.x) {  
    render_basket(&game->basket, (UINT32*)base);  
}
```

PSG Sound -- YM2149 Programmable Sound Generator (`psg.c` , `music.c` , `effects.c`)

All audio is driven by direct register writes to the YM2149 PSG chip at hardware memory addresses `0xFF8800` (select/read) and `0xFF8802` (write). Audio includes:

Background Music (`music.c`): A 48-note looping melody sequenced across PSG channel A. Each note stores a tone divisor value and tick duration. `update_music()` is called once per game loop tick and advances the sequencer when the current note's duration expires.

Sound Effects (`effects.c`):

- Fruit catch -- chime via envelope generator and tone on channel C
- Bomb collision -- noise channel explosion via YM2149 noise register
- Game over -- descending three-step tone melody
- Victory -- ascending tone sequence

All PSG register access enters and exits privileged (supervisor) mode via `Super()` to safely access hardware addresses.

Keyboard Input (`input.c`)

Input is handled non-blocking using `cconis()` (check if key available) and `cnecin()` (read key without echo). At startup the TOS keyboard repeat flag is disabled via a direct privileged memory write so held keys do not flood the input buffer:

```
old_ssp = Super(0);  
*((char *)0x484) = 0; // disable key repeat  
Super(old_ssp);
```

Controls: `A` moves left, `D` moves right, `ESC` quits.

Collision Detection (`model.c`)

AABB (axis-aligned bounding box) collision is computed for both fruit-basket and bomb-basket pairs. Fruit collision deactivates the fruit and increments score. Bomb collision triggers the explosion sound effect, game over melody, sets the quit flag, and exits the game loop.

Dynamic Difficulty (`model.c`)

Elapsed game time is derived from the TOS 70Hz clock: `(get_time() - start_time) / 70`. Spawn intervals for both fruit and bombs decrease and object velocities increase as seconds accumulate, both capped at minimum thresholds to keep the game playable.

```
spawn_rate = 90 - (seconds / 2);  
if (spawn_rate < 30) spawn_rate = 30;
```

68000 Assembly -- Video Base (`v_base.s`)

The video base address is set via a small 68000 Assembly routine that writes the high and mid bytes of the framebuffer address directly to the Atari ST video hardware registers, bypassing the OS entirely.

Game State Model (`model.h`)

The entire game state is held in a single `GameState` struct passed by pointer through every module -- no globals for game data. Entities: `Basket`, `Fruit[40]`, `Bomb[50]`, `Score`. Active/inactive objects are tracked with an `active` flag to allow pooling without dynamic allocation.

Project Structure

<code>main.c</code>	-- Game loop, double buffer swap, TOS clock timing, game over screen
<code>model.c</code>	-- Game state init, collision detection, fruit/bomb spawn and
<code>movement</code>	
<code>renderer.c</code>	-- Per-frame rendering with dirty-rect optimization
<code>RASTER.C</code>	-- Low-level bitmap blitting, pixel ops, screen clear, font rendering
<code>input.c</code>	-- Non-blocking keyboard polling, quit flag management
<code>events.c</code>	-- Game event handling (catch, bomb collision, game over trigger)
<code>psg.c</code>	-- YM2149 PSG chip register read/write via privileged mode
<code>music.c</code>	-- 48-note background music sequencer on PSG channel A
<code>effects.c</code>	-- Sound effects: chime, explosion, game over melody, victory
<code>font.c</code>	-- Bitmap font data table and character rendering
<code>menu.c</code>	-- Splash screen with animated text fade-in
<code>v_base.s</code>	-- 68000 Assembly video base address setter
<code>test.c</code>	-- Hardware test utilities
<code>MAKEFILE</code>	-- Build system using cc68x cross-compiler

Building

You need the **cc68x** cross-compiler toolchain targeting the Motorola 68000.

make

This produces a `game` binary that runs natively on Atari ST hardware or a compatible emulator such as Hatari.

Authors

Tameem Aboueldahab, Andres Isturiz