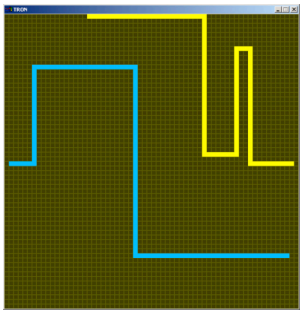
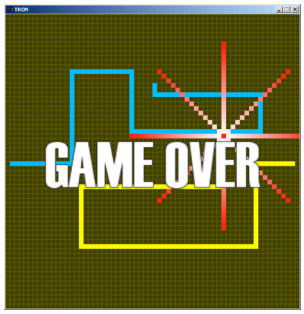
CS371 Spring 2007
Prof. Morgan McGuire

Assigned: Mon. 2/5/07
Due: Mon. 2/12/07 8:00 pm

Project 0: TRON





This project introduces the technology you'll use throughout the semester: the C++ language, G3D library, and iCompile build system on FreeBSD. Most of the information that you need to successfully complete this project is available in the manuals for these tools (e.g., type "man printf" to see the C++ documentation for the printf function). This handout gives you hints that you will need to track down in the manuals. This week only, there is be one lecture entirely devoted to C++ and the scheduled lab time on Thursday is set aside for helping you get started with C++ and Unix programming.

Your goal in this project is to create an implementation of the classic TRON video game for two players seated at the same keyboard, something like the web applet http://kryshen.pp.ru/games/bmtron.html. The specification for this program and implementation tips are later in this document. **The primary challenge of this assignment is learning to use the provided tools and reading the manuals.** The actual game code should be very easy for you to write and extend.

Configure Your Environment

Set the PATH, INCLUDE, and LIBRARY environment variables to reference the cs371 directories in your .cshrc, .tcshrc, .bashrc or other configuration files[1]. If you are using the default CS department configuration and have not changed your shell, you should edit your ~/.local_bashrc file to contain the lines:

```
PATH=$PATH:/usr/local/371/bin
INCLUDE=$INCLUDE:/usr/local/371/include
LIBRARY=$LIBRARY:/usr/local/371/lib
```

You will probably also want to create a web browser bookmark for the G3D and iCompile documentation, which is at:

[1] To do this you'll need to use a unix-style editor like Vi or Emacs. You'll need to use an editor to write your code anyway, so this is a good time to get used to it. I recommend Xemacs, which has a moderately steep learning curve but supports some GUI gestures and will serve you well throughout your career in software development.

file:///usr/local/371/g3d-doc/index.html

Log out and then log back in to ensure that your configuration changes have taken effect.

TRON Specification

The TRON game simulates two "lightcycles" driving at high speed on a plane. The lightcycles build a wall behind them as they move, and crashing into one of these walls destroys a lightcycle.

In the science-fiction TRON movie, which was the first film to incorporate significant 3D graphics, the lightcycles are virtual motorcycles ridden by the main characters and their trails are translucent glowing polygon. In the TRON video game from the same era that you are mimicking, these were represented by single pixels and their trails as lines. Real-time rendering for a game is harder than offline rendering for a movie, so the visual quality is usually lower in games.

Your TRON game must have the following features:

1. Two lightcycles that leave trails
2. Absolute keyboard controls for each lightcycle:
    a. Right side: arrow keys
    b. Left side: W = up, A = left, S = down, D = right
3. Thin lines showing the outlines of grid cells
4. Game ends when a lightcycle hits a trail or outer wall
5. Game ending is followed by:
    a. An animated explosion at the location of the first lighcycle crash, and
    b. The words "Game Over" in Dominant font displayed on screen
6. Lightcycles cannot turn in a direction that will cause them to immediately crash
7. Restart by pressing the spacebar keypress after the game ends
8. ESC to quit the game at any point (this is provided by the default G3D configuration)
9. Written in C++ for FreeBSD using G3D
10. No debugging output in release mode (e.g., turn off debugShowRenderingStats and comment out app->debugPrintfs.)
11. Takes a screenshot whenever the "p" key is pressed (see RenderDevice::screenshot in the documentation.)

For future assignments you may choose to use any language, platform, and libraries you wish, although only C++ and G3D on the department's FreeBSD systems are supported.

Please use Java (yes, Java) coding conventions for your C++ code. The starter main.cpp file already uses these conventions and serves as a guide if you are not familiar with them.

Implementation Tips

Start your program by creating a directory named "tron". In that directory, run iCompile. It will ask if you want it to create a default project set up (which you do want). To make sure that your environment is basically correct, immediately run

```
icompile --run
```

in that directory. This will compile your program and run it. When run, you program print "Hello World!" If this doesn't work, check your configuration and seek help. Do not continue until you have successfully made Hello World compile and run.

Now you're ready to go from "Hello World" to the 3D equivalent[2]. Copy the file

```
/usr/local/371/data/font/console-small.fnt
```

to your "tron/data-files" directory. Then copy the file

```
/usr/local/371/demos/empty/main.cpp
```

into "tron/source". This main.cpp file is a basic G3D program with event handlers already set up for you.

Separate the single main.cpp file into App.h, App.cpp, Demo.h, etc. files so that the interface and implementation are not combined together. Each file .h file should contain a single class definition, a header guard, and include G3D before other libraries. For example,

```
#ifndef APP_H
#define APP_H
#include <G3D/G3DAll.h>

class App {
   …
};

#endif
```

Once the basic program is compiling, strip out the Sky and other 3D rendering and replace them with 2D rendering only.

---

[2] If you have previous Unix experience, you may be doing something clever like connecting to the Unix lab remotely via SSH to work on your project. Stop doing that now. OpenGL programs require an X11 connection, and even if you have set one up correctly, a remote X11 connection will run incredibly slowly and possibly crash.

G3D has several "Image" classes: Image3, Image4, Image3uint8, Image4uint8, Map2D, GImage, and Texture. These all serve different purposes. For this assignment, use an Image4 to represent your game map, where each pixel is one whole block representing the lightcycle or trail in the final image. The "4" in Image4 means that there are four channels per pixel: ARGB. You'll use the alpha value to distinguish an uncolored block from a colored one. Wrapping the Image4 in your own "Map" class would be a good design decision at this point. The reason you'll use Image4 for this assignment (even though GImage would be much more efficient) is that the next several assignments all need features only available on Image3 and Image4, so this is a good time to become accustomed to them.

Every time the Image4 changes (not necessarily every time it is rendered!), convert its pixels from Color4 to Color4uint8 and store them in a GImage. Hen use Texture::fromGImage to turn it into a Texture that can be used for rendering. The default arguments for that method will be too slow for your purposes and will create a blurry map. Specify that the interpolation mode should be NEAREST_NO_MIPMAP and CLAMP should be the wrap mode.

For 2D rendering of the game map to the screen, see RenderDevice::push2D/pop2D and Draw::rect2D. To draw the grid, use RenderDevice::beginPrimitive with RenderDevice::LINES and then issue the 2D vertices corresponding to a set of lines. See the G3D documentation and the OpenGL man pages (beginPrimitive is based on OpenGL's glBegin; sendVertex is based on OpenGL's glVertex.)

The keyboard constants for "normal" keyboard keys all match their lowercase ASCII codes. For example, the following expression is true in the onUserInput method when the "w" key is first pressed:

```
ui->keyPressed('w')
```

Note the single quotes, which indicate a character instead of a string. The arrow key codes are GKey::LEFT, GKey::RIGHT, GKey::UP, and GKey::DOWN.

Use setDesiredFrameRate methods on GApplet to keep your program from running too fast. For responsive user input, process input and render at a relatively high framerate (e.g., 60 frames per second) but have GApplet::onSimulation return abruptly for all but one frame out of four (or ten, or some other number, depending on your implementation and speed.)

Note that both lightcycles can crash at the same time. To be fair to each player, you have to check both when the game ends.

Allocate all variables on the stack or as member variables. Do not explicitly heap allocate. This means that "new" and "malloc" should not appear in the program.

Pass all large (> 8 bytes) objects by reference instead of by pointer (and instead of passing by value, which would just be slow!). For example:

```
// Pass map by reference
void update(Map& map)

// Return a value by reference; only works if
// the return value is a member, not a local variable.
const Map& map() const;
```

Do not use multiple inheritance. It is complicated and misleading in C++.

Remember C++ syntax differences from Java:

- Public, private, and protected are sections of a class definition, not modifiers
- Invoke static methods and members using "::" instead of ".", e.g., Color::red()
- Semi-colon at the end of the class definition, after "}"
- bool instead of boolean
- "const" marks a variable as a constant and a method as non-mutating
- Variables are created on the stack, not the heap, by default
- Methods are non-overridable ("final" in Java terms) by default; "virtual" marks them as overridable

Innovate!

You are invited to go beyond the requirements for this and other projects. Extra work typically takes the form of new features or structuring your code in a particularly good design. Perfectly completing the requirements for each project with zero additional work earns you a B+. To receive an A or an A+ you must exceed the minimum requirements. The real reason to innovate in your projects is not the grade, of course, but instead that graphics is exciting so you'll want to add to your projects.

Some ideas for extending the TRON project include:

- An Artificial Intelligence controller for player 2
- The ability to shoot bullets to slowly break down existing walls
- The end of the trail slowly disappears, so that it moves at ½ the rate of the head
- Make the sides of the world wrap around instead of being walls
- Using additional rendering commands to draw the lightcycles over the map

- Extra cool explosions—try making a "particle system"

Submitting Your Solution

1. Put your name, e-mail address, and the name of the file in a doxygen comment at the top of **each file**, e.g.:

```
/**
 @file Demo.cpp
 @author Morgan McGuire, morgan@cs.williams.edu

 Implements the main applet class for Tron.
 */
```

2. Create a "doc-files" directory that contains a **readme.html** file with your name, e-mail address, partner's name (if you worked in a pair) and anything you'd like to point out to me when I'm reviewing your program. This HTML file must display at least one JPG screenshot (stored in the same doc-files directory) of your program in operation. If there are known bugs, extra credit features, or design points of note, list them here. If you are using your 2-day grace period or a prearranged deadline extension, explain that in the readme file.

3. Make a clean build (using "icompile --clean") and then compile (and test) an **optimized** build of your program using "icompile –O".

4. Change to the **parent** directory of your project and run the FreeBSD command:

```
submit371 tron
```