Modular Digital Game System

An experiment in game design platforms

Interactive Media & Game Development

A Major Qualifying Project Report

Submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Hillary Fotino and Alexander Gray

Advised by Professor Brian Moriarty

ABSTRACT

This project created an Application Programming Interface (API) for a simulated modular digital game system. Each module consists of a triangle that displays colors at the center and edges, monitors an input and signals from the surrounding modules, and communicates with a computer controller. The API allows users to develop game programs for the system. The simulation runs the game files and displays the results. The focus was on practical coding and design of an instructional game system. The challenges in creating such a system provided a valuable learning environment for us in the areas of user interface design, system tool management and design, human computer interaction, and designing educational platforms.

Table of Contents

ABSTRACT 2
TABLE OF CONTENTS ..... **Error! Bookmark not defined.**
INTRODUCTION 4
Problem Statement 4
The Goal 4
The Team 4
DESIGN PROCESS 5
The Simulation 5
Processing 6
Processing.js 6
Design Process 7
The API 8
Javascript 8
Processing 9
Design Process 9
The Hardware 9
DOCUMENTATION 12
Development 12
Implementation 12
PROJECT OUTCOMES 13
CONCLUSIONS 14
APPENDICES 15
Appendix A : References 15
Appendix B: API Code 16
Appendix C: API Documentation 19
Appendix D: Hardware Documentation 20

INTRODUCTION

Problem Statement

Teaching the development of new, unique and interesting games can be difficult, especially when it is very simple to just make a clone of an existing game. Constraining students requires them to be more creative. One method of constraining students to enforce creativity is by giving them an API that is simplified to create very simple games or by giving them a specific hardware set. To accomplish this goal, we set out to create an API for students to use to develop games and a simulation of the hardware for them to test on.

The Goal

The ultimate goal for the hardware was to be a reconfigurable modular system with an API for development and a simulation such that a student could take both pieces and begin to create games even without the hardware component. The intention was also for the hardware to be dynamically reconfigurable into any shape, be it two dimensional or three dimensional. For the purposes of our project, we set out to develop just the API and simulation in full and create a simple prototype of what the hardware could be like. We also chose to limit the simulation to a single, simple 3D solid to test the concept and see if it was viable.

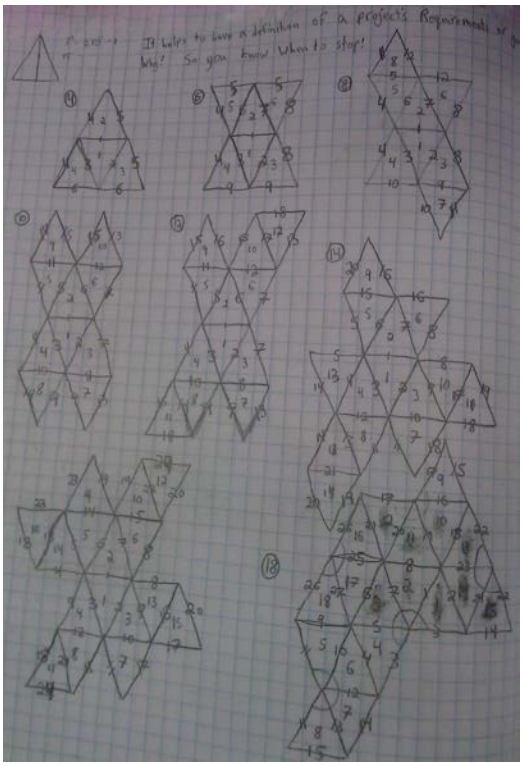
The Team

The project team formed to meet this goal was comprised of two interactive media and game design students, one to handle technical requirements of the software and one to handle artistic requirements and to do the hardware development. Hillary Fotino was responsible for the development of the API, documenting it, and developing the simulation. Alex Gray was

responsible for the creation of the art assets used in the simulation and for developing and building the prototype hardware. He also developed the demonstration Simon game using the API that was developed.

Background

The inspiration for our project came largely from Perlenspiel, a tool for teaching game design which was developed by our advisor, Professor Brian Moriarty. Early in the brainstorming process, we discussed a hardware implementation of Perlenspiel, done with a physical array of Light Emitting Diodes (LEDs). What we eventually set out to make was a system that could be dynamically reconfigured, and was capable of detecting its configuration. This concept is similar to Sifteo Cubes, although they contain Liquid Crystal Display (LCD) screens and can only connect up to six units. Our end goal was to be simpler and more limited from the developer standpoint, but far more configurable from the hardware standpoint.



DESIGN PROCESS

The Simulation

The first step in developing the simulation was to determine which language or framework we would use to create it. The language had to be able to render 3D graphics, preferably from an object file created in a 3D modeling program such as Maya, and need to

have a simple enough syntax that we could pick it up in a short span of time. We narrowed it down to Processing, OpenGL or Processing.js.

Processing

Processing is an open source programming language based on Java. It was developed to be simple way to teach programming in a visual context. Development of Processing began in the spring of 2001 by Ben Fry and Casey Reas while they were students are MIT working at the Media Lab. The development of the project has continued since then, with many different developers working on adding many different libraries to expand the project. Among these are the ability to generate sound, to render 3D objects from object files and networking.

From our perspective, the ability to render 3D objects was the most important ability of the language. The associated library, OBJ Loader, takes the object file, loads it in and renders it to the screen. This allowed us to create our models in Maya and not have to manually develop them in Processing, which could have cost us a lot of development time.

OpenGL

OpenGL was introduced in 1992 and has become a standard for graphical rendering. OpenGL has many useful features, such as hardware acceleration of graphics processing and very extensive documentation, and is an API which has been ported to many different languages include Python, Java, C/C++ and Fortran. However, OpenGL is quite a bit more complicated than Processing or Processing.js, so we ultimately did not choose to use it.

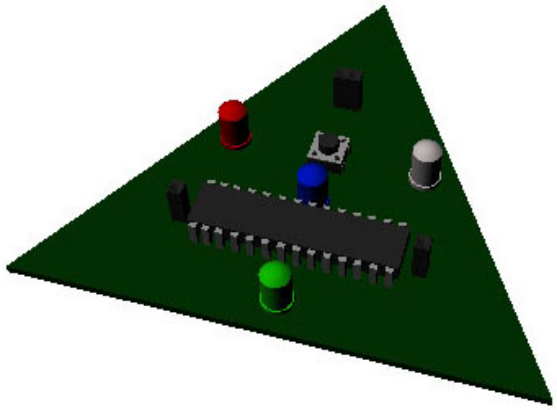
Processing.js

Processing.js is a sister to the Processing project, only instead of being built on the Java language, it was built on the Javascript language. It was developed to make the Processing project more web portable. It is a much newer project than Processing, with the Version 0.4.0 release being made public in early 2010. Because of this, many of the useful libraries that have been developed for Processing have yet to be ported to this project. The object loader libraries

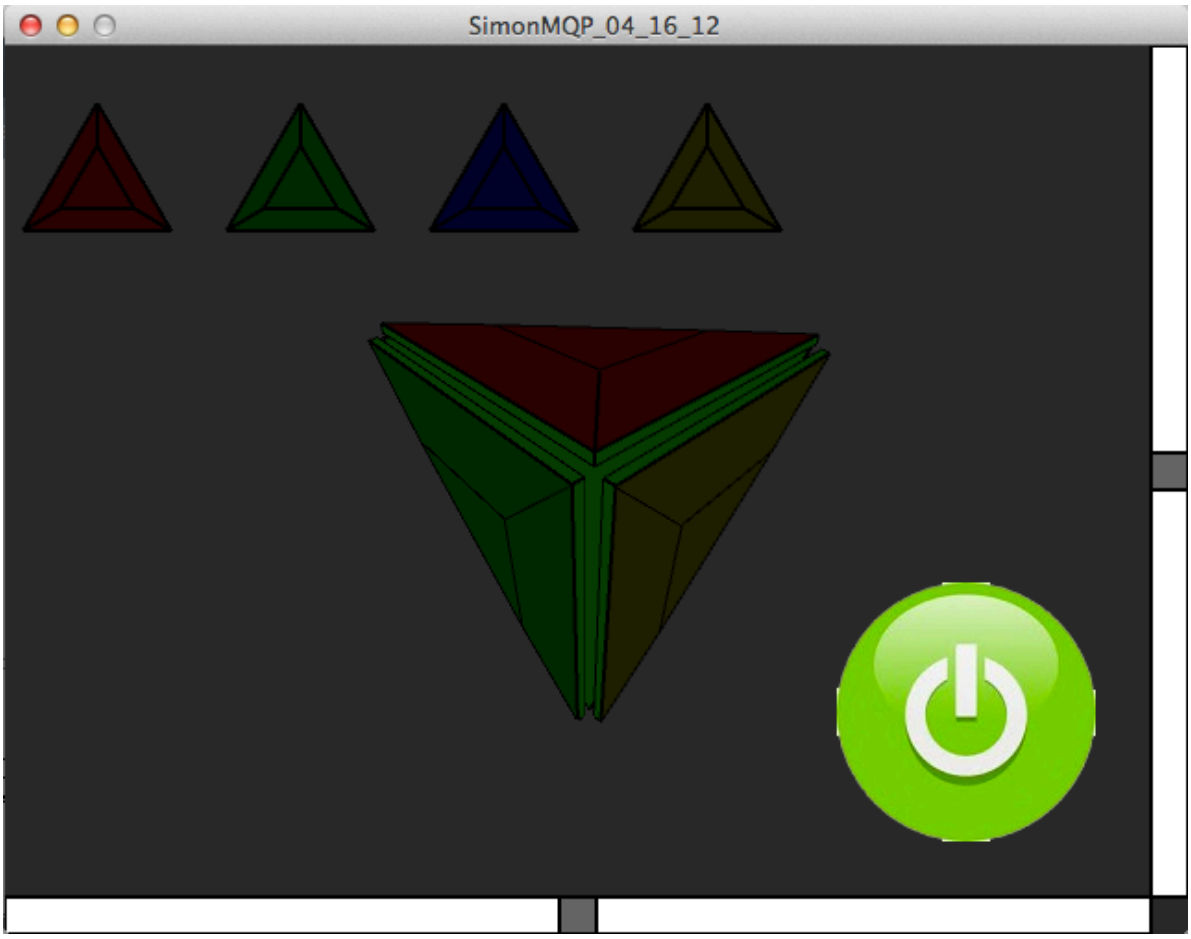
are currently in beta, and while we successfully got access to the beta, the bugs were not hammered out enough to make it a viable choice for our project.

Design Process

Ultimately, we chose to use Processing to develop the simulation. From that point, it was a matter of designing a simplified version of the hardware design as the interface and adding other useful features. The major requirements of the simulation were that it be an accurate simulation that could be used to see how student developed code was working, and that the simulation be able to render the model and allow the student using it to click on the model to simulation pushing the button on that surface. At first, the model being rendered showed all of the electrical components that the hardware would require, and clicking on a surface involved actually clicking on the surface's button.



This was determined to not be a user-friendly interface since it was very easy to miss the small button and showed things that a developer would not care about. We then changed the design to be solid panels of color in place of LEDs and allow the user to click anywhere on a face. Since the model being rendered is 3D and at least one of the faces would always be obstructed from the user's view, we also added icons across the top which represent the faces and react the same way that they do.



The API

The development of the API began with choosing the language to be used. From there, we had to determine the most important functions for developing a game.

Javascript

The original intention had been to develop the API in Javascript so that it would match the Perlenspiel API in simplicity. However, since we chose to develop the simulation in Processing, this meant that Javascript was no longer truly an option. While Processing can be scripted using Javascript, it is not optimal and it runs much more smoothly when written entirely in Processing. The Processing.js project would have allowed for this to happen, by as was stated earlier, does not yet have features that were necessary.

Processing

The API was eventually developed in Processing because it was optimal for the performance of the program. This also allowed the API to be loaded into a given Processing sketch, the name used for Processing programs, as though it was simply part of the project rather than as an external library.

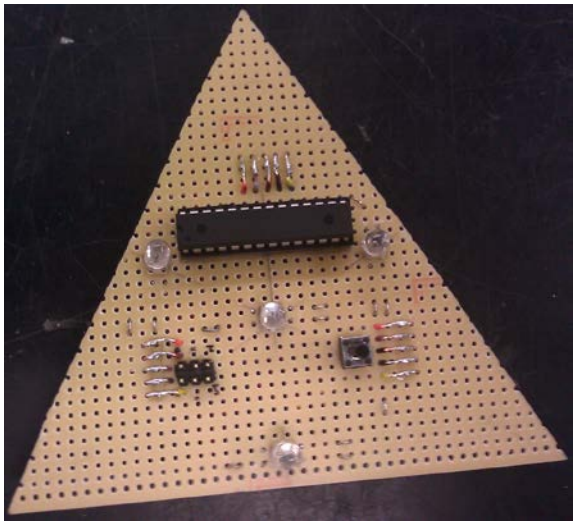
Design Process

When designing the API, the first thing we did was set out the requirements. The requirements for the API were developed by looking through the Perlenspiel API and seeing what kinds of functions and constants it had. Then we compared it to our requirements and constraints. Since Perlenspiel is browser-based rather than being in physical hardware, it had a lot more things it was capable of doing compared to what we needed and could do. We needed to be able to change the colors of the faces of the model, and the subsections of the faces, to handle the button presses on a given face and be loaded onto an Arduino and run. First, a Tetrahedron class was developed which contained all the information about the model, including the port information for the hardware. Then a small selection of helper functions was developed for setting the colors of the faces and face subsections and for reacting to a button press. These were determined to be the major functions that the API required. We chose not to include a speaker on the prototype hardware, so developing sound generation functions for the API was not necessary at this time, though they could be added as a further expansion to the project. The full API can be seen in Appendix B.

The Hardware

In order to make the shape more universally appealing, the prototype boards are composed of equilateral triangles. This lets us make any 3D shape from a Tetrahedron (4-sided solid) to a massive icosahedron (20-sided solid), as well as irregular 3D objects and 2D grids. The modules are designed so that each can be identical, with only one module needing a

connection to power and a programming source. The intention was that additional modules would be connected to the edges of the first module and capable of organizing themselves into a linked system, passing along inputs and outputs to each uniquely addressed board using the Inter-Integrated Circuit (I^2C) communication protocol. The primary output for the face is a set of four Red-Green-Blue (RGB) LEDs, the primary input is a momentary contact push button switch, as shown in the accompanying photograph.



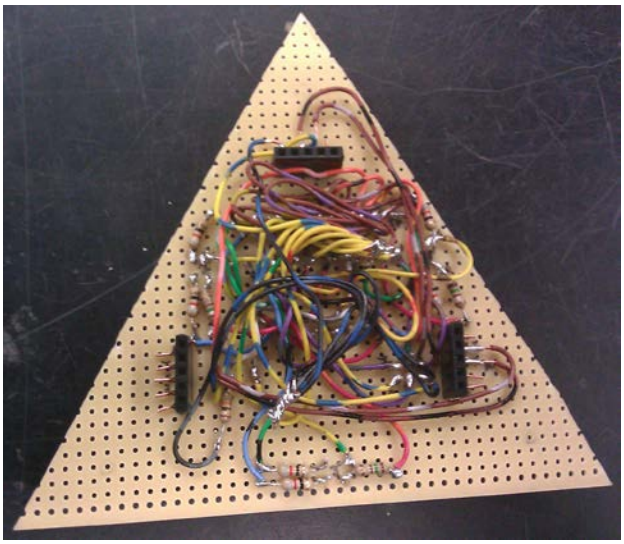
The Microprocessor is an ATtiny88, a 28 pin device that allows for the control of the 12 component LEDs, 1 switch, 3 edge connections, 2 I^2C lines, and 4 pins required to program the chip's logic. There are also 2 pins dedicated to 5 volts and 2 to Ground, leaving 2 additional pins for future use. This chip has programming examples available in C and includes C definitions for the Input/Output (I/O), that makes up the basis of the device code. An Arduino would be coded with a modified C/C++ to be used as the I^2C Master controller and be connected to a computer to transfer commands. Once the Master is connected to the first module the module would be set into a search mode and would then send a signal to its first edge and look for a responding module. The responding module would be addressed as module two and it would wait for module one to check the other edges for other modules. When module one has finished assigning its edges it would tell module two if it found module three and four, then module two would check its two other edges and if the modules there didn't already have an address it would assign them and report what it found back to module one. Module one would then send search commands through to the other modules in the system to establish the overall shape of the system.

The actual implementation of the I²C protocol ended up being more of a challenge to implement on the ATtiny88 chips than expected. As of the project's conclusion, complete hardware functionality is not complete. The prototype tetrahedron's modules are not communicating with each other, the Arduino or the API.

Along the development process the components and dimensions of the system changed as new insights came to light. The thought that the boards could be cut with edge lengths of 2 inches was lost with the realization that the chip footprint, LED, and switch placements would be forced to overlap, and that there wouldn't be enough time or money available to have the circuit boards fabricated using a printed circuit board service.

The next board size used edge lengths of 3 inches and made use of both sides of the board for wiring to allow the central LED and switch to be located properly with accordance to the design. This prototype brought the problem of the interconnect ports for the boards to light. The original thought was to have short wires bent between edge mounted connectors. The tight angles in the tetrahedron's geometry cause the connectors to interfere with each other at this scale.

The problems with the interconnect ports are still present in the 4 inch version due to the lengths of the wiring and the added precautions of current limiting resistors. The figure to the right shows the rear of a 4-inch prototype module.



The design works for well for the purpose of the prototype at this scale. The core functionality is partially implemented, while the software that would run it is currently incomplete.

DOCUMENTATION

Development

When any API is developed, the most important thing that can be developed along with it is proper documentation. It is only with the documentation that it can truly be used to create. The documentation for the digital modular game system was based largely on the documentation for Perlenspiel. This seemed like an appropriate organization given our inspiration for the project, and since the Perlenspiel API is very well organized and user friendly.

Implementation

The documentation itself is done through inline and block commenting in the API file itself. This allows the user to not only read through what a function is supposed to do, but also allows them to look at the function and see exactly how it is behaving. This prevents black box coding problems from occurring on the user end. This also makes it so that the user can look up the documentation without having to go outside of their own program. A secondary documentation set containing all constants and function calls necessary exists and has been included in the appendices as appendix C.

PROJECT OUTCOMES

While we successfully developed the API and simulation for our digital modular game system, and successfully developed a prototype of the hardware, many of the goals we had early on had to be removed from the project due to time constraints. Our ultimate goal to develop something that was fully reconfigurable and dynamic was instead realized as a single tetrahedron with an API and a simulation that fully supported it. The intention shifted from developing a fully functional game system, to prototyping a concept and seeing if it was viable.

The major reason that we kept running into problems that were sufficiently troublesome as to cause us to scale back parts of the project was lack of technical knowledge about hardware development. Future projects working with this, or any, hardware should include an electrical or robotics engineer since they have the background in working with these kinds of systems.

Given the goal of creating an API, simulation and hardware prototype to see if the concept of using a simple hardware setup to create games and game-like toys is an interesting and useful one, we believe that our project has been a success. We developed a prototype for what we set out to make and successfully created a game for it. The creation of the game was fairly simple.

CONCLUSIONS

Looking back over this project, we had many setbacks and successes. We developed more than we set out to make, though less than the vision we had in the beginning. The API and simulation have been tested through the development of a game, and prototype hardware has been produced. While the hardware does not have all of the features that it optimally would have, and is not yet running code developed with the API, it does demonstrate that the concept is viable.

APPENDICES

Appendix A : References

API
http://www.perlenspiel.org/
http://processing.org/
http://opengl.org/
http://processingjs.org/

Hardware
http://www.mouser.com/ProductDetail/Atmel/ATTINY88-PU/?qs=sGAEpiMZZMvu0Nwh4cA1wRKJzS2Lmyk%252bS1XFHRQy1EA%3d

http://www.ladyada.net/learn/avr/whatisit.html

http://forums.adafruit.com/viewtopic.php?f=19&t=16260

http://www.evilmadscientist.com/article.php/avrtargetboards

http://www.instructables.com/id/I2C_Bus_for_ATtiny_and_ATmega/

http://www.expresspcb.com/ExpressPCBHtm/Tips.htm

http://www.edaboard.com/entry861.html

http://www.ermicro.com/blog/?p=1971

http://www.ermicro.com/blog/?p=1239

http://www.ermicro.com/blog/?p=744

http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=27233&start=0

http://iamsuhasm.wordpress.com/tutsproj/avr-gcc-tutorial/

Processing
http://www.openprocessing.org/visuals/?visualID=9397

http://www.openprocessing.org/visuals/?visualID=48874

http://www.openprocessing.org/visuals/?visualID=47549

http://thequietvoid.com/client/objloader/#examples

http://www.openprocessing.org/sketch/7881

http://processingjs.org/articles/jsQuickStart.html

Appendix B: API Code

User Code File:

```
class YourCode{
      YourCode(){
            //constructor
      }
      void lose(){
            println("You LOSE!");
      }
      void start(){

      }
}
```

API File:

```
// game.pde for Modular Game Development

/*
Copyright © 2012 Worcester Polytechnic Institute.
*/

// This is a template for creating new games
// All of the functions below MUST exist, or the engine will stop and
complain!
final color RED = color(256, 0, 0);
final color DARKRED = color(40, 0,0);
final color GREEN = color(0, 256, 0);
final color DARKGREEN = color(0, 40, 0);
final color BLUE = color (0, 0, 256);
final color DARKBLUE = color (0,0,40);
final color BLACK = color (0,0,0);
final color PURPLE = color(256, 0, 256);
final color YELLOW = color(255, 255, 0);
final color DARKYELLOW = color(0x20, 0x20, 0);
final color ORANGE = color(256, 100, 0);
final color AQUA = color(0, 100, 100);
final color WHITE = color(255,255,255);

class Game{
  int id;
  int data;
  int state;
```

```
void Game(){
  state = 5;

}
// Init ()
// Initializes the game
// This function normally includes a call to ModelType(Tetra)
void Waiting()
{
  //does nothing but wait for activation by 's' key
  println("WAITING");
}
void Init()
{
    // change to the dimensions you want

    //ModelType(Tetra);
     println("in init");
     state = 1;// sends game.state to menu
     //simon = new Simon();
     // Put any other init code here
}

// Menu
// displays the game menu and options
void Menu()
{
  //println("in Menu");
  state = 2;// sends game.state to run
}

// Run
void Run(){
  println("in Run");
  //Put code here that you want to run
}

// Lose
void Lose(){
  //Put code here for when the player loses
}

// Win
void Win(){
  //Put code here for when the player wins
```

```
  }


  // Click (id, data)
  // This function is called whenever a board is clicked
  // It doesn't have to do anything
  // id = the board reporting the click
  // data = the data value associated with this board, 0 if none has
been set

  void Click(int id, int data)
  {

  }

  // Release (id, data)
  // This function is called whenever a mouse (switch) button is
released over a board
  // It doesn't have to do anything
  // id = the id of the board that is being released
  // data = the data value associated with this board, 0 if none has
been set

  void Release(int id, int data)
  {

     // Put code here for when the mouse (switch) button is released
over a board
  }

  // Tick ()
  // This function is called on every clock tick
  // if a timer has been activated with a call to Timer()
  // It doesn't have to do anything
  void Tick()
  {
        println("tick");
    // Put code here to handle clock ticks
  }
}
```

Appendix C: API Documentation

CONSTANTS:

Colors:

RED
DARKRED
GREEN
DARKGREEN
BLUE
DARKBLUE
BLACK
PURPLE
YELLOW
DARKYELLOW
ORANGE
AQUA
WHITE

Faces are identified by their number 1,2,3 or 4

Sub face sections are identified as 11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34, 41, 42, 43, and 44.

Functions

void turnOnWholeFace(int faceNum)

Takes the number associated with a whole face and turns all of the lights on.

void turnOffWholeFace(int faceNum)

Takes the number associated with a whole face and turns all of the lights off.

void turnOnSubFace(int subFaceNum)

Takes the number of a sub-face and turns the light on.

void setWholeFaceColor(int faceNum, color c)

Takes the number of a face and a color and sets the color of that whole face.

void setSubFaceColor(int faceNum, color c)

Takes the number of a sub face and a color and sets the color of that sub face.

Appendix D: Hardware Documentation

ATtiny88 pin layout

used for	Pin #	label	used for	Pin #	label
ICSP	1	(PCINT14 / RESET) PC6	I2C	28	(PCINT13 / SCL / ADC5) PC5
LED 1 R	2	(PCINT16) PD0	I2C	27	(PCINT12 / SDA / ADC4) PC4
LED 1 G	3	(PCINT17) PD1	Edge 3 Comm.	26	(PCINT11 / ADC3) PC3
LED 1 B	4	(PCINT18 / INT0) PD2	Edge 2 Comm.	25	(PCINT10 / ADC2) PC2
Push button switch	5	(PCINT19 / INT1) PD3	Edge 1 Comm.	24	(PCINT9 / ACD1) PC1
LED 2 R	6	(PCINT20 / T0) PD4	Expansion Pin	23	(PCINT8 / ADC0) PC0
+5 volts	7	VCC	Ground	22	GND
Ground	8	GND	Expansion Pin	21	(PCINT15)
LED 2 G	9	(PCINT6 / CLKI) PB6	+5 volts	20	AVCC
LED 2 B	10	(PCINT7) PB7	ICSP	19	(PCINT5 / SCK) PB5
LED 3 R	11	(PCINT21 / T1) PD5	ICSP	18	(PCINT4 / MISO) PB4
LED 3 G	12	(PCINT22 . AIN0) PD6	ICSP	17	(PCINT3 / MOSI) PB3
LED 3 B	13	(PCINT23 / AIN1) PD7	LED 4 B	16	(PCINT2 / OC1B / SS) PB2
LED 4 R	14	(PCINT0 / CLK0 / ICP1) PB0	LED 4 G	15	(PCINT1 / OC1A) PB1