Author Topic: Prototype Whitewood Project (Read 23738 times)

Steve S
FPGA_testers
Posts: 434
Steve Shoyer

Re: Prototype Whitewood Project
« **Reply #30 on:** October 17, 2012, 11:57:57 PM »

I'm not sure how your cabinet is designed, but most playfields that I've seen will have a notch on the lower left side of the playfield, opposite the shooter lane, so that there's clearance for the left flipper switch.

--Steve

Edited to change right to left, doh!

« Last Edit: October 18, 2012, 11:46:50 AM by Steve S »

Logged

JoeShabadu2000
Wizard
Posts: 118

Re: Prototype Whitewood Project
« **Reply #31 on:** October 18, 2012, 10:25:05 AM »

Steve, there is a notch already on the lower right side for the shooter, I assume you mean lower left there. All the pinball machines I own have the notch on the lower left side of the playfield like you are describing, but all of my machines have a left outlane kickback mechanism and I assumed that the notch was just there for the kickback. I can see though that there might be an issue with clearance for the the left flipper switch, it's probably not a bad idea to notch it out just to be safe. I'm not sure this playfield will be going in to a cabinet anyway, I am working on the assumption that it won't be perfect the first time through so I am envisioning 2nd and 3rd revisions in my future. Thanks for the input, if you notice anything else that looks weird please let me know!

Logged

Steve S
FPGA_testers
Posts: 434
Steve Shoyer

Re: Prototype Whitewood Project
« **Reply #32 on:** October 18, 2012, 11:53:15 AM »

Yes, left rather than right. I think the kickback mechanisms mount onto the playfied, so it might need a shallow pocket so the plunger is at the same plane as the ball, and perhaps a hole for wiring or for the coil lugs.

Logged

JoeShabadu2000

Wizard

Posts: 118

Re: Prototype Whitewood Project
« **Reply #33 on:** November 05, 2012, 10:25:27 AM »

Just as a quick update, I feel like I am very close to actually running the full playfield through the CNC router. Getting the correct measurements for the plastic playfield inserts took me a lot longer than it really should have, but I am pretty confident now that all is good. Now the issue is finding a time when I can get 4-5 free hours on the CNC router at the local makerspace. I will post more details and photos once I actually get it done.

In the meantime, I wanted to post a few things from my good friend Andrew Lundin. I have been talking with him about how to approach the programming side, since in my opinion he is really an ace coder, even though he doesn't have a lot of experience with Python. I have posted before about my relative lack of programming experience, so Andrew graciously wrote some stuff up for me about how to approach object oriented programming in general, as well as some thoughts about how to approach implementing one of the rules that is going to be in my game. After reading his emails, I came away with a better understanding of the whole process, and I thought that others on this board might benefit as well from his thoughts. I got his permission to repost them here. Keep in mind that Andrew doesn't have a P-ROC and has limited experience in Python, but if you are as confused as I was then this might give you a place to start.

You mentioned that you might ask for someone to code the rules for you, or you might give it a shot yourself. I briefly considered trying to build an intermediary system to allow you to define and manipulate the rules through a simpler syntax, or perhaps through a GUI, but you really NEED the power of a full-blown programming language to accomplish a lot of what you want to do, and Python is a great choice for that... FWIW, I think you should at least give it a try, because I'm pretty certain you'll take to it like a natural-born coder. As a creative problem-solving process, it can be a really fun challenge to figure how to make the program do what you want, and I think you'll find it very gratifying as you progressively solve those problems and end up with something that is wholly your own creation.

I'd like to offer you some tips on how to implement your rules. In the most general sense, this is a pretty straightforward state-machine. You'd have a set of variables for global state (game mode(s)) and the states of each component. How each component responds to input events would depend on these state variables. If I were building this from scratch, I'd define a set of objects to represent each category of component, and a "singleton" object to represent the entire game. I'd build an event-messaging queue, which would be polled in a loop by the game object, to dispatch events to any components that need to know about them, which would then respond to each event according to all the state variables. This object-oriented, event-driven architecture would be quite a challenge to set up in the first place, but it would make all the rest of the coding vastly simpler. Actually, now that I think about it, I recall you're using a framework specifically designed for implementing pinball rules -- I'd wager they've already provided most of those basics...

...Yep. I just read up on PyProcGame, and it looks like they did it pretty much exactly as I would have, and they took it several steps further. Well... Alright, then. Awesome. With this framework, it should be relatively easy. So I'll just give you a quick run-down of some important concepts to bring you up to speed. I apologize if any of this is already familiar to you.

Object-oriented programming is a strategy of problem decomposition which encapsulates variables and functions into functional sub-units that can be defined independently from the rest of the program. This "separation of concerns" makes *it much easier to design program behavior by breaking it down into more manageable chunks that interact in (hopefully) intuitive ways. OO code is easier to maintain, because its complexity is structurally separated into simpler parts, which makes it easier to understand when you go back and read it after you may have forgotten some details (or if someone else reads it). It's also easier to re-use OO code, because it's designed in a modular way.*

But this kind of power comes with the trade-off of some conceptual overhead that isn't necessarily easy to wrap your brain around. You probably already know that a "class" defines a TYPE of object. Classes can have "properties" (member variables) and "methods" (member functions). Just as a variable is an abstraction of a value, a class is another layer of abstraction on top of that, defining a consistent structure of variables, and the functions that act on them. Similarly, just as it is meaningless to attempt to read the value of a variable before it has been initialized, in the same way, a class doesn't usually do much by itself until you "instantiate" it -- create an object that is an "instance" of that class. (That's actually the definition of the word "object" : an instance of a class.) This is usually done by calling its "constructor", which is a special method that creates and returns a new instance of that class. (In Python, the constructor is called "__init__", but it's different in other languages.) You can have any number of instances of a class (unless it's a "singleton" class, which means it's specifically designed to prevent more than one instantiation). Each instance has its own unique set of values for each of its defined properties. Methods often read and manipulate the values of instance properties, so when you call a method on an object, the variables it's working with are unique that that particular instance.

For example, you could have a "Player" class with a property for "score". This class might have a method like "change_score(points)" which would simply do "score += points". You CAN'T call the change_score method on the Player class directly, because the class itself does NOT contain an actual variable for score; it only defines that each of its instances should have such a variable. But you could create any number of Player instances, each with its own independent score. Calling the change_score method on a certain Player instance would only affect the score of whichever instance it's called on. As a side note, there are also "static" properties and methods which apply to the whole class, independent of any particular instance, but they're fairly rare.

The "scope" of a variable determines where you can access it from, which is determined by where it's defined. If you define a variable within a function, that variable only exists within that function, and you can't access it from anywhere else. Function arguments are variables defined in the function declaration which receive the data passed in by whatever called the function, and their scope is likewise limited to that function. A variable defined in a class is a property, which belongs to an instance of that class. You can access properties (and methods) of an object from outside that object, but you have to reference the instance with the "." operator, as I'm sure you've seen many times. Many OO languages have "access modifier" keywords that allow you to declare a property or method as "private", meaning it can only be accessed from within the same class, or "public", meaning it can be accessed by any code that has a reference to it (more on references later). Any variable NOT defined within a function or class has "global" scope, meaning it can be accessed from anywhere. In some languages, globals are automatically available in every function, but in some languages, you have to explicitly declare which global variables you want to access in a particular function, using the "global" keyword. Some languages have other scoping constructs, such as packages, modules, namespaces, and others. The main goal of limited scope is to allow the re-use of variable names without any ambiguity about what you're actually accessing. For example, you might have a "Game" object with a method that tallies up the points earned during a bonus round or something. You might define a variable called "score" within that function, but there's no confusion that you might be accidentally accessing the "score" *property of a Player instance, because their scope doesn't overlap at all -- they're totally separate variables that happen to have the same name. Whenever the scope does overlap between variables with the same name, most languages provide a way to clarify which one you're referring to -- I'll get into that later.*

It's tempting to make a lot of your variables global, so you don't have to pass them to every function that needs them, but don't get carried away with that, because then you can't use those variable names anywhere else. Any globals you do define should have very unique names that are highly unlikely to be used elsewhere. Many programmers generally frown on the use of globals, because they muddle up the modular separation of concerns that makes OOP so powerful. But globals can provide a very helpful simplification if you're not trying to build some megalithic application that NEEDS such clean modular separation. Personally, I tend to use globals pretty often in the early stages of development, and then eventually migrate most of them into a more appropriately limited scope.

You don't have to define every object from scratch. You can take another class as a starting point and add more properties and methods onto it, creating a "subclass". Subclasses "inherit" (contain and have access to) all the same properties and methods of their base class (or "super" class, or sometimes "parent" class), and can re-define ("override") methods already defined in the base class, while still being able to access the original versions of those methods through a bit of extra syntax. So whenever there are categorical similarities between multiple classes, you can abstract their common elements into a base class, from which each of the others can be derived. This eliminates redundant code and allows you to make any changes in a single place, instead of editing every similar class. The syntax of subclassing varies widely from language to language, but in Python the class declaration looks like this:

```
  class SUBCLASS_NAME(BASE_CLASS_NAME)
```

Subclasses can be nested in a hierarchy, meaning you can subclass a subclass. Some languages allow "multiple inheritance", meaning that classes can be derived from multiple base classes simultaneously, inheriting ALL of their properties and methods, but this can lead to some headaches with namespace collisions, causing some ambiguity when two or more base classes have the same property or method. Another strategy similar to multiple inheritance, but with fewer drawbacks, is the use of "interfaces", which define that any class which implements the interface must implement certain methods which take certain arguments and return certain types of return values. This provides a sort of communications protocol between classes, so that you don't necessarily need to know anything about what a class is doing internally, as long as you know it correctly implements a particular interface, so you can call any of the interface's required methods and rest assured that it will respond as expected. In some languages, interfaces are simply a design pattern accomplished THROUGH multiple inheritance (which is why C/C++ code usually puts class and function declarations in header files, separate from their definitions -- the actual procedural code they contain -- because the declaration IS the interface). But in some other languages, an "interface" is a language-defined construct provided in lieu of multiple inheritance, to basically enforce this design pattern. Python DOES allow multiple inheritance, and does NOT have any defined "interface" construct. To disambiguate members with the same name inherited from more than one base class, Python simply uses the one that comes first in the list of base classes. While not always ideal, this solution does keeps it all perfectly clear, which is a good thing. Concepts related to subclassing and inheritance include virtual functions and abstract base classes, but that's getting into some advanced territory you don't necessarily need to dive into yet.

Logged

JoeShabadu2000
Wizard
●●●●●
Posts: 118

Re: Prototype Whitewood Project
« **Reply #34 on:** November 05, 2012, 10:26:35 AM »

References are another important concept to understand. When you call a function that takes an argument that is a simple data type, such as a number or a string, its VALUE is what gets passed into the function's argument, even if you used a variable to pass it. Inside the function, that argument is handled as a totally separate variable, so you can change its value without affecting the variable you passed into the function call. In contrast, objects are usually passed "by reference", which isn't a unique copy of the original, it's just an address to the actual object itself. So when a function receives an object reference in an argument, it can make permanent changes to the original object, which persist even after the function returns. Some languages allow you to specify whether to pass an argument by value or by reference. This can be useful if you want to modify the original variable passed in, which is one way to return more information than a single return value. There is some variation between different languages regarding default argument passing conventions, so that's an important point to familiarize yourself with when you learn a new language. Here is an in-depth discussion about the way Python handles this:

http://stackoverflow.com/questions/986006/python-how-do-i-pass-a-variable-by-reference

In some OO languages, references to instance properties and methods are implicit when used within the same class. As I mentioned, non-static methods execute in the context of a particular instance, so all class members for that instance can be automatically included in the scope of all methods of that class, meaning you can simply refer to the name of a class member without any additional syntax, and it will automatically be interpreted as belonging to the instance on which the method was called. Many OO languages provide the keyword "this" as a placeholder for the reference to the current instance, to allow you to clarify the context of the reference. Even when member reference is implicit, the keyword "this" can be used to disambiguate a class member from a global of the same name, or from an argument of the same name. For example, the Player class might have a method called "set_score(score)". In that method you'd have to refer to the class property as "this.score", to clarify the member reference as opposed to the argument reference, because the argument takes precedence, because its scope is more local. So, to set the score property to the value passed into the score argument, you'd write "this.score = score". In some languages, you are required to specify the instance reference along with every member reference, even within a method of the same class.

Python is a little weird on this point. All member references must include an instance reference, but Python doesn't provide any built-in keyword to do that, such as "this". Instead, Python requires that all methods must be declared with an instance reference as the first argument to the method. The instance reference argument can be given any name, but by convention it is usually named "self". Then, within that method, you can refer to a class member as "self.score" or whatever. When the method is called on a particular instance, Python automatically passes the instance reference under that first argument's name. But when you call the method, you omit that first argument, since Python handles that automatically. This can be confusing, because the method declaration has more arguments than the method call. Most other languages avoid such inconsistencies, except where arguments may be optional or of indeterminate quantity, so this quirk of Python is fairly controversial. I hope I've saved you some frustration by explaining that.

In many high-level languages, including Python, a function is a "first-class" *object, meaning you can store it in a variable and pass it to other functions, just like any other data type. Of course, functions (including methods) are handled by reference (including instance reference). So when you pass a function as an argument to another function, you're giving the called function access to call the passed function, which is often called a "callback" function. (Say THAT sentence 3 times really fast!) To put that another way, when you pass a callback function, you're requesting that the callback should be called either right away or at a later time, when some condition is met, as defined in the other function you pass it into. This provides an elegant solution to asynchronous event handling, which you'll be doing a lot of in your pinball rules implementation. You can run into something tricky with callbacks: you should only pass the NAME of the function, without any parentheses or arguments. The name is the reference. Whenever a function reference is followed by parentheses, this initiates an actual CALL of that function, replacing itself with the return value of the function, instead of passing a reference to the function.*

Python is a dynamic language, which means that class and function definitions can be changed at any time. You can add a new method to a class defined elsewhere, or replace a function with a different version of it on the fly. This also allows you to define "anonymous callback" functions directly within another function call. This is called a "closure", and it executes in the context of the calling code, meaning that any variable defined in the same scope as the code that passed the closure to another function will also be accessible within the closure itself. These dynamic features are extremely powerful, but can also be very confusing, and since it's beyond the scope of what you need to know to get started, I won't go into it any further here.

One more thing I'll add, that isn't particular to OOP, but you'll need to understand to use PyProcGame, is an architectural design element called "hooks". A hook is similar to a callback, in that you expect it to be called under certain conditions, but you don't pass its reference. Instead, a special naming scheme is used to allow the framework to recognize that a particular function is a hook of a certain type or purpose. Under hook-based frameworks, the code is scanned to search for functions with names matching the pattern of the hook naming scheme, and any hooks it finds are indexed by the framework so that they can be called when the conditions for triggering them are met. As an example specific to PyProcGame, you can define a method called "sw_mySwitch_active", which will be called whenever the switch "mySwitch" changes to its active state. I don't think the PyProcGame documentation ever mentions the term "hook", but that's what this is.

Well, that's all I can think of for now. Jeez, I didn't mean to write a book! I guess I should have just referred you to an "Intro to Object Oriented Programming" article or something. I'll bet Wikipedia has a pretty good outline of it. Anyway, there's the "Drew's Notes (TM)" on the subject. I've glossed over and omitted several points, but if you can understand and remember most of the concepts I described, then you've got a pretty solid foundation for learning any OO language, and you're ready to dive into the PyProcGame documentation. If you persevere in this, I have no doubt that you'll become a proficient programmer, and I'm sure you'll be using that skill quite a bit in the years to come.

Logged

JoeShabadu2000
Wizard
Posts: 118

Re: Prototype Whitewood Project
« **Reply #35 on:** November 05, 2012, 10:28:13 AM »

For this next bit, Andrew has provided some sample code to implement one of my game rules, which is a player operated magnet save on the left outlane. The player needs to light the M-A-G targets on the playfield, which then allows for the magnet save to be activated when the player pushes a button on the left side of the cabinet.

Alright, here's some sample code. Let's just jump right into some pyprocgame code to implement one of your rules:

```
import procgame

class MAG_Waiting_Mode(procgame.game.Mode):
  def __init__(self, game):
    super(MAG_Waiting_Mode, self).__init__(game=game, priority=1)

  def sw_MAGtargetM_active(self, sw):
    self.game.lamps.MAGtargetM.enable()
    self.M_enabled = true
    self.check_MAG_complete()
    return procgame.game.SwitchStop

  def sw_MAGtargetA_active(self, sw):
    self.game.lamps.MAGtargetA.enable()
    self.A_enabled = true
    self.check_MAG_complete()
    return procgame.game.SwitchStop

  def sw_MAGtargetG_active(self, sw):
    self.game.lamps.MAGtargetG.enable()
    self.G_enabled = true
    self.check_MAG_complete()
    return procgame.game.SwitchStop

  def check_MAG_complete(self):
    if self.M_enabled and self.A_enabled and self.G_enabled:
      mag_enabled_mode = MAG_Enabled_Mode(self.game)
      self.game.modes.add(mag_enabled_mode)


class MAG_Enabled_Mode(procgame.game.Mode):
  def __init__(self, game):
    super(MAG_Enabled_Mode, self).__init__(game=game, priority=1)

  def sw_MAGSaveButton_active(self, sw):
    self.game.coils.MAGSaveCoil.pulse(500)
    return procgame.game.SwitchStop


class JoeShabaduGame(procgame.game.GameController):
  def __init__(self, machine_type):
    super(JoeShabaduGame, self).__init__(machine_type)
    self.load_config('joeshabadugame.yaml')

  def reset(self):
    super(JoeShabaduGame, self).reset()
    first_mode = MAG_Waiting_Mode(self)
    self.modes.add(first_mode)
    self.enable_flippers(enable=True)


game = JoeShabaduGame(machine_type='wpc')
game.reset()
game.run_loop()
```

This code is incomplete and untested, so don't expect it to actually work (although it MIGHT, sort of). You'll have to define the various switches, lamps and coils in your YAML file:

```
switches: MAGtargetM, MAGtargetA, MAGtargetG, MAGSaveButton
lamps: MAGtargetM, MAGtargetA, MAGtargetG
coils: MAGSaveCoil
```

Note that I've used the same name for the switch and lamp of each MAG target. I believe this should be okay, because they're stored separately, and because the pyprocgame example code does the same thing with "startButton". But feel free to rename them however you like -- just be sure you change every occurrence of them in the code.

I'll run through this program in the same sequence as the machine would. Class and function definitions are simply stored for later use, so we'll proceed to the first statements that can be executed immediately:

```
game = JoeShabaduGame(machine_type='wpc')
game.reset()
game.run_loop()
```

We create a variable called game, and assign to it the return value of the constructor of the JoeShabaduGame class. In calling the constructor, Python first creates an instance of that class, and then calls the __init__ method of the class, automatically passing the instance reference in the "self" argument. The first line of __init__ is a little cryptic:

```
super(JoeShabaduGame, self).__init__(machine_type)
```

Every instance of a subclass also contains instances of all its base classes. The "super" function returns a reference to the base class, GameController, from which JoeShabaduGame was subclassed. When you call the "super" function, you have to specify which subclass and instance you're referencing to get the base class instance of this particular subclass instance. (That isn't necessary in most OOP languages -- I don't know why Python can't figure it out.) Then we call the __init__ method of that base class instance, which does all the dirty work to build the GameController object for us.

Then we load and process the YAML config file. The "load_config" method is defined in the base class, but take note that in this case we don't need to use the "super" function to reference a member of the base class. By inheritance, all members of the base class become part of the subclass, so you can access them directly through "self.whatever". But it's also possible to "override" methods of the base class by defining a new method of the same name in the subclass, which effectively replaces the base class method of that name. That was the case with the __init__ method, which was the reason we needed to call the "super" function, to bypass the override that would normally take precedence otherwise. (It doesn't matter that we happen to be within the overridden __init__ already, because functions can call themselves recursively.)

The __init__ method completes, and the constructor returns the instance reference to our newly created JoeShabaduGame object, which we store in the "game" variable. Then we call a couple of methods to set it up and get it running. The "reset" method is defined in the subclass, overriding the base class method of the same name. Inside that method, the first thing we do is to call the original base class method, through the "super" function. I have no idea *what the base class "reset" method does internally, nor do we need to right now. Then we create a new instance of the MAG_Waiting_Mode class, subclassed from Mode. Its constructor simply calls the constructor of the base class, and returns the instance reference, which we store in the "first_mode" variable. JoeShabaduGame's base class, GameController, contains a "ModeQueue" object, which stores a reference to every active mode and handles the prioritized event dispatching to them. So, to make the MAG_Waiting_Mode active, we simply add it to the ModeQueue, which is stored in the "modes" property of the base class, and inherited by the subclass. The flippers are enabled, and the reset method returns.*

All we have left to do is to start up the event handling loop, which polls for input events and dispatches them to each mode according to their respective priority. Once we call the inherited run_loop method on the JoeShabaduGame object, it will continue to wait for events and handle them one at a time, indefinitely.

Now, let's say the ball hits the G target, activating the MAGtargetG switch defined in the YAML config file. This generates an event, which the ModeQueue will dispatch to the only mode that is currently active, MAG_Waiting_Mode. When we first instantiated this mode (back in the JoeShabaduGame reset method), the pyprocgame framework scanned its code looking for any methods that match the naming scheme of event handlers (sw_SWITCHNAME_active), and it should have found:

sw_MAGtargetM_active
sw_MAGtargetA_active
sw_MAGtargetG_active

and stored a reference to each of them for later use. Most event handling systems require you to call an "add_listener" method to explicitly listen for events, and you must pass it a reference to your event handler callback method, but under pyprocgame's hooks system, we don't have to do anything else to be able to receive events, other than give the event handler method a special name, which is pretty freaking cool.

Now that we have an event for the MAGtargetG switch, the ModeQueue will automatically call our event handler to allow the MAG_Waiting_Mode to respond to this event however we like. So, inside the sw_MAGtargetG_active method, the first thing we do is to turn on the MAGtargetG lamp. Note how we access this lamp: self.game.lamps.MAGtargetG.enable(). When the MAG_Waiting_Mode was instantiated, we passed it a reference to the JoeShabaduGame object (which was called "self" at the time, because we were inside the reset method of that class). The Mode base class of MAG_Waiting_Mode stored this reference in a property called "game". So, within any method of the MAG_Waiting_Mode class, we can access the JoeShabaduGame object through "self.game". The GameController base class of JoeShabaduGame contains a property called "lamps", which stores a collection of every lamp defined in the YAML. To access a specific lamp, we first have to access the GameController, then its lamps property, then the lamp name, as defined in the config file. So, within the MAG_Waiting_Mode class, "self.game.lamps.MAGtargetG" will access an instance of the Driver class, which represents a lamp, coil, solenoid, flasher, etc., and this particular instance represents the MAGtargetG lamp. The Driver class provides a method called "enable", which enables the driver indefinitely, turning the lamp on until we explicitly turn it off later.

The next thing we do in our event handler is to create a property called "G_enabled" within the MAG_Waiting_Mode object, and set its value to "true". This is our state variable representing the fact that the G target has already been hit, which we need to keep track of to be able to determine when the M-A-G objective has been completed. In this simple example, it wasn't strictly necessary to create this property, because we could have retrieved the state of the lamp through the Driver's "state" method when we check for completion of the objective. But what if you want to make the target lamps flash repeatedly *prior to being hit? It's better to have a separate variable to represent the abstract state of the target, rather than relying on the state of its lamp.*

Once we've set the G_enabled state variable to true, we need to check whether the M-A-G objective has been completed, because the other two targets may or may not have already been hit. I defined a method of the MAG_Waiting_Mode class called "check_MAG_complete" to handle this. I could have included this simple conditional test within each of the switch event handlers, but that would have been redundant. So we call check_MAG_complete, which tests the value of each of the state variables to determine if they are all true. Conditional expressions are handled with boolean logic, and here I've used the "and" operator to require that all three state variables must be true for the entire expression to be true, otherwise the check_MAG_complete method does nothing.

When all three targets have been hit, check_MAG_complete can finally enable the MAG Save button. We do this by instantiating a new Mode object, MAG_Enabled_Mode, passing it the reference to the game object, and then adding this new mode to the ModeQueue to make it active, as described previously. Note the differences from the last time we did this, which happened within the "reset" method of the JoeShabaduGame class. Note how the "game" object reference is passed to the mode's constructor upon instantiation, and note how we access the ModeQueue ("modes") within the MAG_Waiting_Mode class. Do you understand these differences?

Once added to the ModeQueue, MAG_Enabled_Mode is ready to receive just one event: the activation of the MAGSaveButton. When this event occurs, the ModeQueue calls the sw_MAGSaveButton_active method, which pulses the MAGSaveCoil for half a second. The documentation said that you can blow a fuse if you leave a coil on too long. I guess this duration is something you'll have to tune.

You may notice that I didn't remove the MAG_Waiting_Mode from the ModeQueue, so it's still in there, receiving events whenever the MAG targets are hit, creating a brand new MAG_Enabled_Mode object every time, which will stack up in the queue indefinitely, each one pulsing the coil for an additional half a second every time the MAGSaveButton is pressed, so this would be a serious problem in a real program. I also didn't define any way to disable the MAG Save once it has been enabled, and reset the MAG objective to its initial state, mostly because you didn't define the conditions under which that would occur. I'll leave these tasks to you as an exercise.

Python is pretty weird to me. The code seems cleaner, in some ways -- I like the use of block indentation to eliminate the need for enclosing brackets, because code blocks should always be indented anyway. But the requirement of the instance reference in every method declaration just bugs the crap out of me. I've never encountered another language with that requirement before, and it seems like completely unnecessary and counter-intuitive clutter to me. But I'm sure there must be a good reason for it...

Anyway, one last disclaimer: As I've mentioned, I have very little experience with Python. Some of the explanations I've given above are based on some educated guess-work, so if an experienced Python developer tells you something that contradicts what I've said, you should probably take their word for it.

« Last Edit: November 14, 2012, 08:27:02 AM by JoeShabadu2000 » Logged

JoeShabadu2000
Wizard

Re: Prototype Whitewood Project
« **Reply #36 on:** November 05, 2012, 02:38:50 PM »

Posts: 118

Since I am in a posting mood at the moment, I wanted to talk a little more about CAD/CAM and how all this stuff works in practice. Before I started working with this stuff myself I really didn't have a good handle on the whole process, so I figure there are other people confused as well. As always, I am not claiming to be an expert on any of this stuff, I'm just learning as I go, if any of my concepts don't sound right here please let me know.

Let's start at the "end", as it were, with the CNC rotuer itself. Basically, the CNC router consists of a rotating spindle that contains a bit to cut the material (in my case a standard router), and some associated motors and electronics that let you position the router in 3 dimensional space (x, y, and z axes). CNC routers use stepper motors, which turn a defined amount every time that a pulse is applied to them. Once the machine has been calibrated to determine how large of a distance each step moves the router, you can tell the machine where to go in physical space to a very accurate degree.

You need a CNC controller to interface with the motor drivers, one popular example (and what is used on the CNC I have access to) is called Mach 3. Mach 3 takes a text file, written in a language called GCode, and translates that into the pulses necessary to make the stepper motors position the router in the location that the program specifies.

Gcode itself isn't too hard to understand. As I learned from rapper 50 Cent, the first rule of GCode is no snitching. Oh sorry, that is something else. Here is a quick example of some GCode with an explanation that follows:

Code: [Select]

```
( 1-16 in mill Drill 3-8 in deep )
( Mach2/3 Postprocessor )
N20G00G20G17G20G90G40G49G80
N30G70
N40T4M06
N50G00G43Z1.2362H4
N60S3000M03
N70G94
N80X0.0000Y0.0000F25.0
N90G00X7.6075Y4.7098Z0.2362
N100G01Z-0.3750F3.0
N110G00Z0.2362
N120G00X8.3575
N130G01Z-0.3750F3.0
N140G00Z0.2362
N150G00X7.6075Y1.1473
```

The purpose of this code is to drill 4 holes, each one 0.375" deep. The "N" at the start of each line is just the line number, starting at 10 and ending at 240 in this example. The commands in the example above generally start with a G or an M. Here is a link to a quick reference about what the G commands mean in the context of Mach 3. G0 is fast move, G1 is linear move, etc. Starting on N90, you can see that we are moving to a certain set of X and Y coordinates, with a Z coordinate that is about 1/4 in off the surface. In N100, we move to the Z coordinate of -0.375, which will drill the hole. In N110, we move the Z axis back up to above the board so that we can move the tool, then in N120 we move to another set of X and Y coordinates.

For a simple example like this, it doesn't seem too far fetched to think that you could just open up your text editor and bang something like this out. However, once you start dealing with programs of any complexity, it starts to get very overwhelming very quickly. This is where CAD and CAM come in.

CAD is where you do your design work. Once you have your design completed and in layers (more on that later), you save it to a standard format (such as DXF) so that it can be imported into the CAM program. The CAM program is what takes your CAD design and translates it into GCode that can be understood by the CNC controller software.

The CAM program that I have access to is called Cut2D, produced by Vectric. This software costs $150. CAMBAM is another low cost option I have heard about, but I haven't used that one. There are also some free CAM programs, but in my limited experimentation I wasn't really able to get much use from them, so I think this may be an area where you just want to pony out the dough if you need this kind of program. In general, if you are using someone else's CNC router, they are going to have some sort of CAM program set up that has settings tuned for that particular CNC. There are many applications, Cut2D included, that also include a CAD portion, so you can design and export GCode from the same program. For the purposes of milling a pinball playfield, you should be able to get by with a 2D CAM program, with a few workarounds that you may have to do here and there. 3D CAM programs can be significantly more expensive, and for the most part there just isn't a need for those extra capabilities for these purposes.

Generally, the CAM program is going to be concerned with Tools and Operations. Your CAM software will be set up with a list of end mills and other bits that are available for your router, and it will use the properties of these bits when it creates a toolpath to cut out your design. Different CNC routers have different properties for feed rates, etc., and different tools have different properties about their pass depth and things of that sort. Assuming that your CNC router does not have an automatic tool changer (very few hobbyist class machines do), you will need to generate separate GCode for each different tool that you are using. So, if you have some operations that use a 1/4" mill and some that use a 1/8" mill, you will need two separate files, and you will load the second file after changing the tool manually.

Operations that you have available are going to depend on the software that you are using. In the case of Cut2D, the operations are Drill, Profile, and Pocket. Drill simply moves the bit down and back up, so if you are making a hole the same size as your drill bit you are all set. Profile traces a line around the edge of the shape you have specified, so for example a Profile operation on a ball trough hole would make a cut around the outside and leave you with a nice chunk of wood from the center to remove yourself. A Pocket operation uses the mill to cut out the entire section of an object, staring from the inside out. So a Pocket operation on a ball trough hole would leave the same cutout as the Profile operation, but there would be no chunk of wood in the middle to remove. The pocket operation will also take significantly longer that the profile operation to complete, since the mill is having to perform so many more moves to complete the operation. And there is a tremendous amount of additional dust created. You need to use Pocket operations on any cavities that need to be created that don't go all the way through the playfield, such as the magnet coils that sit underneath the playfield.

Since the CAM software is concerned with Tools and Operations, in my opinion it is easiest to import files into your CAM software with these criteria already in mind. For simple projects, it may be easy to remember which tools and which operations need to be performed for each object, in which case you can just import the whole project at once. However, for something like a full playfield, it is easy to get lost in what tools go with which object, and how deep the various holes need to be. At minimum, I would at least split the CAD file by tool type, although you may want to split the files by operation as well, that way there can be no confusion about what needs to be done in the CAM software to generate the proper GCode.

Which leads us to the CAD portion. I have been using DraftSight from Dassault Systems, which is a free and very powerful 2D drafting program. My understanding is that it works almost exactly like AutoCAD, so you are really getting a professional level product at no cost. Dassault also makes SolidWorks, which is basically the industry standard 3D CAD program, so I think they are hoping you might upgrade to SolidWorks at some point if you need to move away from 2D. It also costs $4,000. I think you are going to be good with DraftSight, though.

The way I think about my design starts with Blocks. A Block is one component in the machine, be it a pop bumper, or a flipper assembly, or an insert or post. Each Block is drawn up in its own individual file, and then these Blocks will be imported into your playfield design file. The cool thing about using blocks is that if you need to update a measurement or something like that for a particular block, you just edit the block itself, then re-import the block into your design and overwrite the existing block definition. All blocks of that type will update automatically based on the changes you have made. This has been important to me with my plastic inserts in particular, since I have had to change their dimensions several times over the course of my design. Blocks are also nice because they allow you to move, rotate, etc. a whole collection of related objects at one time. With a flipper assembly for example, you just grab the block and rotate it, instead of having to select each hole, flipper, bracket, etc. individually.

So, when you are setting up the blocks that will go in to your playfield, think ahead about how that is eventually going to interact with the CAM software. The way I have conceived of this is by thinking about 5 general "zones" (not really a recognized CAD term but that is how I think of them), each of which has its own color in the CAD file.

Items that take up space on the bottom of the playfield (like lamp brackets, magnet brackets, etc.) are drawn in blue in my examples. I call this zone "Bottom". From the CAM program's perspective, these bottom items are irrelevant since they don't directly affect what happens to the playfield, but you need to have the correct dimensions so that you can be sure that you don't have brackets that overlap or that sort of thing. I use a dashed line for this color to help remind me that it is not visible from the top of the playfield.

Moving up, my second zone are items that need to be worked in to the bottom of the playfield, but do not go all the way through. This includes things like pilot holes and pockets for magnets, and in my drawings I have this zone in cyan. I am also using a dashed line for this color. I call this "Bottompilot".

The middle zone I have in green, and is for holes that need to be made all the way through the playfield. These holes could be made either from the top of the playfield or the bottom, it shouldn't matter as long as you have the orientation correct. This color and the ones that follow are all solid lines, so that I know they are visible from the top of the playfield. I call this zone "Hole".

Moving up is the magenta zone, for holes that are made from the top of the playfield but do not go all the way through. I call this "TopPilot".

The final zone is the white zone, for items that sit on top of the playfield. Like the bottom zone, this zone doesn't affect the CAM program, but we need to make sure that nothing overlaps and that we have enough room for our ball to travel. This one I call "Top".

So, thinking in a general way about these zones will help us when we are setting up our "layers", which is a recognized CAD term and one that you should be familiar with. Your drawing can have as many layers as you want, and you can name each layer a different thing, give them different colors, different line styles, etc. In naming my layers, I use the following convention:

Zone Name - Bit Name (Operation and Depth) - Hole Size (if a regular hole)

So, for a 3/4" hole that needs to be made in the playfield, I would use something like:

Hole - 1-4 in mill (Pocket Through) - 3-4 in

In my case, 1/4" is the biggest mill I am using, so any holes with a diameter of 1/4" or larger are going to use this tool. I am using the Pocket operation to make sure I don't have any cores floating around, if I used Profile here then there would be a small chunk of MDF in the middle that wouldn't get milled, and might get tossed around by the bit. The Through descriptor lets me know that I need to make the final depth of cut slightly larger than the thickness of the playfield. I put a separate descriptor for regular holes to let me know their diameter, in some cases you may need to change all holes of a certain diameter to be a different size, if you have them all grouped together on the same layer it makes it easy to view and change them as a group.

The idea is that you want to use the same naming convention for each layer as you are making your blocks. Then, when you have inserted multiple blocks that use the same layer names, the objects from the different blocks that have the same layer name are combined together. When you get to the end of your design, it should be (relatively) easy to save individual layers (or groups of layers) to their own individual CAD files for import in to your CAM program.

I think that is all I have for now, I will post some more when I can actually get around to getting this first whitewood all ready to go.

Logged

JoeShabadu2000

Wizard

Posts: 118

Re: Prototype Whitewood Project
« **Reply #37 on:** November 14, 2012, 08:28:48 AM »

Just a few quick updates from Andrew after he read through the thread again.

I wanted to point out that the "game" property of the Mode class is NOT the same as the global variable "game" defined at the end, if that wasn't already obvious, nor is it the same as the argument "game" taken by the constructor (__init__) of the Mode subclasses. Incidentally, they all happen to contain the exact same data (the instance reference to the JoeShabaduGame singleton object), however they are completely separate variables that simply have the same name in different scopes, so that's why they're accessed in different ways. In addition, the Mode and GameController classes provided by the framework are defined within the namespace "procgame.game", so you have to refer to those base classes in the context of that namespace when you derive a subclass from them (see the subclass declarations). That "game" namespace is a totally different thing that is NOT a reference to the JoeShabaduGame object -- it's not even a variable, it's just a scoping construct (unless Python treats namespaces as objects, too, which wouldn't surprise me). In practice, you shouldn't have to worry about technicalities like that, as long as you follow the conventions of the sample code. But it's important to understand this when you reuse the same names in different scopes, whether or not they hold the same data (because they certainly don't have to).

One other thing, I use the word "argument" where others may use the word "parameter" -- they are synonymous and interchangeable. The latter term seems more prevalent in Python documentation.

Logged

JoeShabadu2000

Wizard

Posts: 118

Re: Prototype Whitewood Project
« **Reply #38 on:** December 04, 2012, 08:03:29 PM »

I have been having some CNC issues, which have troubled me for the past few weeks. Hopefully I will be able to get over these problems soon!

I am using a CNC router at my local "hackerspace", which is great because I don't have to build and house my own CNC router, and the pricing is very reasonable. The downsides are that it is certainly less convenient to use than the tools at my house, and as I get in to it, it appears that the CNC router is not as high quality as I would like, and this is affecting my ability to get this playfield done.

The first issue I had is that the work area of this machine is about 24"x30", which is not enough to do a full playfield (20.5"x46" standard). To get around this limitation, I have to route the upper half and lower half of the playfield separately. This means generating separate CAD and GCode sets, which is a pain but not too big of a deal. The larger problem is getting everything lined up properly, so that the holes I am making on the lower half line up with the holes on the upper half. Certainly I am not the first to have this problem, and most of the solutions I have found involve things like registration holes, and making sure that the work piece is exactly square and exactly the right size. The theory behind registration holes is, you drill some holes through the work piece into the spoilboard, that are mirrored across both the lengthwise and width dimensions of the work piece. In my example, these would be maybe 1/4" from each edge of the playfield (mirrored across the Y axis), and at even intervals from the middle of the long axis of the playfield (which is at 23", so you might have holes at 21" and 25"). You would then put some pins through these holes and route your lower half. Then, you remove the pins, rotate the playfield so the upper half is in the cutting area, and replace the pins. If all your measurements are correct, and everything is perfectly square, then everything should line up exactly. This system should also allow you to flip the playfield upside down, so that you can drill holes in both the bottom and the top surfaces of the playfield, and have those holes line up.

Unfortunately, as I have recently discovered, the CNC router I am using is not exactly square. You can confirm this by programming the router to cut a square into a piece of wood. In my case, I made a square that was 20.5" per side. When measuring the diagonals, I discovered that one of the diagonals was 3/16" longer than the other. Also, when I placed my MDF work piece (which I understand comes perfectly square from the factory) onto this square and lined up one edge of the MDF with one edge of the routed square, the other edge of the routed square diverged from the other edge of the MDF.

I had some clues about this earlier in the process. When I did some tests of my registration hole system, I noticed that it was significantly more difficult to replace my pins once I had rotated the work piece. I thought that this was based on my not having the work perfectly square in relation to the router, but it turns out this was not the case necessarily.

Although this is unfortunate, I feel like I can get around these issues, at least enough to get a prototype created. I won't be able to drill pilot holes in the top of the board and have them line up with the bottom, but there are only a few of those in my design and I will do them with a hand drill. The only other thing I need to cut in the top is the openings for the inserts, and for those it is less critical that the placement be exact, as long as everything is within 1/16" or so. I have also arranged my design so that there are no elements that cross the center line for the upper and lower halves, so if there are slight alignment issues it shouldn't affect things too much.

The other issue I have is with the electronics on the CNC, which is a more vexing issue. As other people have mentioned, it takes a long time for the router to make all these holes (6 hours, in my case), so for it is only really practical to do this on the weekend. The other important thing to understand is that the program controlling the router doesn't have any sensors to detect the position of the router, it only knows where it has told the router to go, and assumes that the router went where it was told. Also, if the machine makes any wrong cuts, you are pretty much screwed and have to go back to square one with a fresh blank.

The first time I tried to route the full playfield, I had a strange error where it appeared the machine was inverting the Y axis commands as soon as it got to the point where it was routing the hole for my ball trough. So instead of looking like a front slash (/), it looked like a back slash (\), and that messed up all the other holes I had cut for my flipper assemblies, etc. The good news (?) is that this happened relatively early in the process, so I only wasted a couple hours.

The second time I tried the full playfield, I separated out the code for the trough so I could cut that separately, and everything went fine there. In fact, everything was going fine for several hours, until I was on my very last program of GCode, which was the bottom half of the top layer, cutting the last holes for the inserts. I went to the other room briefly, and heard an unfamiliar noise come from the stepper motors. When I got back, the router was still cutting, but in the wrong place in the X and Y axes. What I believe happened is that the electronics controlling the stepper motors didn't properly relay some of the commands, possibly due to overheating or something like that. I guess it also could have something to do with the stepper motors themselves, but that seems less likely to me. In either case, unfortunately my playfield was ruined, and 6 hours down the drain.

I'm going to give it another shot this weekend, I will let it "cool down" for a while between jobs this time so hopefully the electronics don't crap out on me. Reading on CNC forums and such, apparently this is a problem that can happen when you don't use high quality electronic components for the motor drivers. And, the router not being square is a common problem that people have when building their own CNC routers, so I guess that shouldn't be surprising either. I am hopeful that I will be able to make it work with what I have, but I know Steve from these forums has had to build his own CNC router, so I may have to give that some more serious consideration if I continue to have issues here.

I have attached a few photos of the underside, which I think looks great, and of the top, which is ruined. The joys of DIY!

IMAG0035.jpg (55.79 kB, 892x534 - viewed 427 times.)

IMAG0036.jpg (52.96 kB, 892x534 - viewed 393 times.)

IMAG0037.jpg (63.56 kB, 892x534 - viewed 407 times.)

IMAG0038.jpg (48.2 kB, 892x534 - viewed 404 times.)

Logged

Steve S
FPGA_testers
Posts: 434
Steve Shoyer

Re: Prototype Whitewood Project
« **Reply #39 on:** December 05, 2012, 12:13:57 AM »

Most of the CNC machines I've seen use stepper motors to move the spindle, so there's no feedback to let the machine know where it is. Too bad we couldn't combine a CNC with a P^3 playfield so it could track the cutter's position.

If you've got a playfield that's mostly OK but has a couple of mistakes, you could fill the holes with Bondo or wood filler and recut the holes that were missed. It'll save you a lot of time (and wood), especially if you're still in the early stages of testing the playfield layout. You can even do that on purpose if you're fine tuning and just want to reposition a few components. I've got a target bank planned that will need a lot of tweaking, so I'll probably cut a large (4"x6" or so) hole and use inserts to test different target configurations.

Logged

ironspider
Wizard

Posts: 106
Replay Design Studio

Re: Prototype Whitewood Project
« **Reply #40 on:** January 21, 2013, 04:26:05 PM »

Just got started on thinking about my first custom game and I feel like this thread is my going to be driving the way for me! Any updates since December Joe?

EDIT: Changed October to December!
« Last Edit: January 22, 2013, 10:30:33 AM by ironspider »

Logged

JoeShabadu2000
Wizard
Posts: 118

Re: Prototype Whitewood Project
« **Reply #41 on:** January 22, 2013, 11:22:33 AM »

Glad to hear that the thread has been of some use to you. Unfortunately I am still stuck in the CNC router phase, the one that I had been doing my testing on and was planning on using for the playfield doesn't seem to be up to the challenge at the moment. I really can't proceed much further without having a completed whitewood playfield, and I'm not sure at this point how I am going to accomplish that. The other side of things is that I am very busy with my job for the next few months, so I may be in a holding pattern for a bit regardless. I have some ads up on Craigslist to see if there is a CNC hobbyist or something with some better equipment in my area that would let me use their machine for a little while. My other thought is to build my own CNC router, which I know Steve from the forums has done, but a machine that will suit my needs is going to be really expensive and I'm not sure if it is worth dropping the coin for something that I may not use that much.

Logged

ironspider

Wizard

Posts: 106

Replay Design Studio

Re: Prototype Whitewood Project
« **Reply #42 on:** January 22, 2013, 11:40:41 AM »

Thanks for the reply Joe, I'm sure I'm not the only one following this thread!

Bummer on the C&C holdup. I have a lot of woodworking experience and after trying to drill 6 countersunk holes, the same depth, in a straight line, on my HyperPin cabinet I was like "Wow, never again." I think I might get lucky in that there's a brand new MakerWorks place in my city that appears to have a 48"x96" router.

Of course I have, literally, no understanding of CAD or automated routing or any of that stuff so I can't wait to get frustrated trying to marginally learn something there! 🙂 Just like I can't wait to realize I know nothing about Python!

I'm sure I'm months behind your current location so you've got time 🙂 j/k. Please keep us updated, even if it's just with your frustrations, so my inner child stays excited.

Logged

JoeShabadu2000

Wizard

Posts: 118

Re: Prototype Whitewood Project
« **Reply #43 on:** January 22, 2013, 02:12:06 PM »

I also recently received the following question from "J" via email, I thought I would share with the group:

I am just walking through your thread for about the 3rd time and I was curious about something as it relates to the Power Supply. My question is whether or not any part of the P-ROC system actually uses 12v? There are some AnTek power supplies that provide 70c/24v/5v(regulated) for about $40 cheaper than the P-ROC guys sell them for. If the P-ROC doesn't actually use 12v then I could just molex up all my power from the AnTek supply and not have to mess with the ATX PSU at all. I plan to use 5v for LEDs lamps.

I may be wrong about this, so I'm going to post it on the board in hopes that someone will correct me if so. The 12V input on the P-ROC is used to generate the voltage for the switch matrix. There is also a jumper on the board that allows you to switch from 12V (Williams) voltage to 5V (Stern) voltage used with the matrix, so I initially thought that you wouldn't need a 12V connector if you were using 5V for the switch matrix. However, according to section 4.12 on the board specs, it looks like the 12V is regulated down to 5V when the Stern switch matrix is checked, so I think you are still going to need a 12V power section.

That being said, you might still be able to get by without a separate ATX power supply. On some other projects, I have used Linear Voltage Regulator ICs to drop a higher voltage down to a lower one. If you search eBay or Google for "12V regulator IC" you will find a bunch of items that will be able to take the 24V from the power supply and convert it down to the 12V used by the P-ROC. The current on most of these is limited to 1 Amp, but my guess is that the 12V on the P-ROC wouldn't draw nearly that much.

Most people already have PC that they are using to control the P-ROC, though, so having an ATX power supply hook up is no big deal for most people. Are you planning on using an embedded controller or something of that sort?

Logged

ironspider

Wizard

Posts: 106

Replay Design Studio

Re: Prototype Whitewood Project

« **Reply #44 on:** January 22, 2013, 03:00:32 PM »

Hey Joe, I was just curious sinec Antek offers a 70/24/5 PSU and thought if 12v wasn't used then we could possibly get Gerry to switch to that in the starter kit. It is, in fact, used for exactly what you said 🙂

I'm just going to use ATX (for the 5v/12v) as I start out since I know, like probably all of you guys, I have 20 of these things laying around 🙂

Quote from: JoeShabadu2000 on January 22, 2013, 02:12:06 PM

> I also recently received the following question from "J" via email, I thought I would share with the group:
>
> *I am just walking through your thread for about the 3rd time and I was curious about something as it relates to the Power Supply. My question is whether or not any part of the P-ROC system actually uses 12v? There are some AnTek power supplies that provide 70c/24v/5v(regulated) for about $40 cheaper than the P-ROC guys sell them for. If the P-ROC doesn't actually use 12v then I could just molex up all my power from the AnTek supply and not have to mess with the ATX PSU at all. I plan to use 5v for LEDs lamps.*
>
> I may be wrong about this, so I'm going to post it on the board in hopes that someone will correct me if so. The 12V input on the P-ROC is used to generate the voltage for the switch matrix. There is also a jumper on the board that allows you to switch from 12V (Williams) voltage to 5V (Stern) voltage used with the matrix, so I initially thought that you wouldn't need a 12V connector if you were using 5V for the switch matrix. However, according to section 4.12 on the board specs, it looks like the 12V is regulated down to 5V when the Stern switch matrix is checked, so I think you are still going to need a 12V power section.
>
> That being said, you might still be able to get by without a separate ATX power supply. On some other projects, I have used Linear Voltage Regulator ICs to drop a higher voltage down to a lower one. If you search eBay or Google for "12V regulator IC" you will find a bunch of items that will be able to take the 24V from the power supply and convert it down to the 12V used by the P-ROC. The current on most of these is limited to 1 Amp, but my guess is that the 12V on the P-ROC wouldn't draw nearly that much.
>
> Most people already have PC that they are using to control the P-ROC, though, so having an ATX power supply hook up is no big deal for most people. Are you planning on using an embedded controller or something of that sort?

Logged