Adding auto lights to chisel

- propose to add two options to `txt2pen.py`
  - `-l` to enable auto lights
  - `-f num` to change the default frequency of lights (default is every five squares)

Code changes to chisel/python/txt2pen.py

```
inputFile = None
defines = {}
verbose = False
debugging = False
autoLights = False
floor = []
rooms = {}
maxx, maxy = 0, 0
doorValue, wallValue, emptyValue = 0, -1, -2
versionNumber = 0.1
lightFrequency = 5
```

- notice the new global variables `autoLights` and `lightFrequency`

Code changes to chisel/python/txt2pen.py

```
def usage (code):
    print "Usage: txt2pen [-dhlvV] [-f frequency] [-o outp
    print "  -d debugging"
    print "  -h help"
    print "  -l automatic lighting"
    print "  -f frequency    (every frequency squares pla
    print "  -V verbose"
    print "  -v version"
    print "  -o outputfile name"
    sys.exit (code)
```

Code changes to chisel/python/txt2pen.py

```
class roomInfo:
    def __init__ (self, w, d):
        self.walls = w
        self.doors = d
        self.doorLeadsTo = []
        self.monsters = []
        self.weapons = []
        self.ammo = []
        self.lights = []
        self.autoLights = []
        self.worldspawn = []
```


Code changes to chisel/python/txt2pen.py

```
def handleOptions ():
    global debugging, verbose, outputName, autoLights, li

    outputName = None
    try:
        optlist, l = getopt.getopt(sys.argv[1:], ':df:hlo
        for opt in optlist:
            if opt[0] == '-d':
                debugging = True
            elif opt[0] == '-h':
                usage (0)
            elif opt[0] == '-l':
                autoLights = True
            elif opt[0] == '-f':
                lightFrequency = int (opt[1])
            elif opt[0] == '-o':
                outputName = opt[1]
etc...
```


New function checkLight

```
def checkLight (p, l, lightCount):
    if lightCount == lightFrequency:
        l += [p]
        lightCount = 0
    else:
        lightCount += 1
    return l, lightCount
```

- which is called from your `introduceLights`

txt2pen changes

```
def generateRoom (r, p, mapGrid, start, i):
    global verbose, rooms, debugging

    if verbose:
        print "room", r,
    p = moveBy (p, [-1, -1], mapGrid)
    if verbose:
        print "top left is", p
    s = p
    walls, doors = scanRoom (s, p, mapGrid, [], [])
    if debugging:
        print walls
    rooms[r] = roomInfo (walls, doors)
    rooms[r].autoLights += introduceLights (s, p, mapGrid
```

function printRoom changes

```
etc...
    o = printMonsters (rooms[r].monsters, o)
    o = printAmmo (rooms[r].ammo, o)
    o = printWeapons (rooms[r].weapons, o)
    if autoLights and (rooms[r].lights == []):
        o = printLights (rooms[r].autoLights, o)
    else:
        o = printLights (rooms[r].lights, o)
    o = printSpawnPlayer (rooms[r].worldspawn, o)
    o.write ("END\n\n")
    return o
```


function printRoom changes

- you need to complete `introduceLights` to make these changes take effect

pen2map

- `chisel/python/pen2map.py`
  - `pen2map` converts a `pen` file into a `map` file (doom3)

pen2map

```
$ cd $HOME/Sandpit/chisel/python
$ python pen2map.py -h
Usage: pen2map [-c filename.ss] [-dhmtvV] [-o outputfile]
  -c filename.ss    use filename.ss as the defaults for t
  -d                debugging
  -e                provide comments in the map file
  -g type           game type.  The type must be 'single'
  -h                help
  -m                create a doom3 map file from the pen
  -s                generate statistics about the map fil
  -t                create a txt file from the pen file
  -V                generate verbose information
  -v                print the version
  -o outputfile     place output into outputfile
```

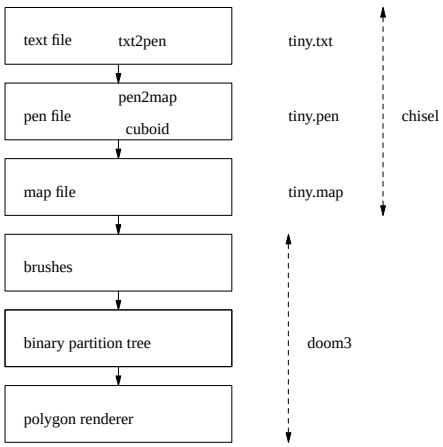
pen2map overview

- parses a pen file, creates internal data structures representing the pen map
  - it then iterates over the rooms and generates a doom3 map file
- conceptually the generation of the rooms is rather like virtualised lego (within chisel)
  - pen2map generates blocks and places these blocks into a world
  - it will attempt to join blocks together as long as this results in a bigger cuboid structure
- however the doom3 map uses planes and not blocks!

Construct the program in logical units

- Henry Legard proverb
- one way to achieve this is to layer the solution
  - divide and conquer
- consider our doom3 tools

Doom3 and chisel layering



Minimal box defined in the map format

```
brushDef3
{
  // floor of fbrick
  (0 0 -1 0) ((0.0078125 0 0.5) (0 -0.0078125 -1)) "textures/hell/cbrick2b" 0 0
  // ceiling of fbrick
  (0 0 1 -288) ((0.0078125 0 0.5) (0 -0.0078125 -1)) "textures/hell/cbrick2b" 0
  // top most horizontal of fbrick
  (-1 0 0 -480) ((0.0078125 0 0.5) (0 -0.0078125 -1)) "textures/hell/cbrick2b" 0
  // left most vertical of fbrick
  (0 -1 0 -576) ((0.0078125 0 0.5) (0 -0.0078125 -1)) "textures/hell/cbrick2b" 0
  // bottom most horizontal of fbrick
  (1 0 0 432) ((0.0078125 0 0.5) (0 -0.0078125 -1)) "textures/hell/cbrick2b" 0 0
  // right most vertical of fbrick
  (0 1 0 528) ((0.0078125 0 0.5) (0 -0.0078125 -1)) "textures/hell/cbrick2b" 0 0
}
```

- six planes which define a cuboid

The second plane

- is the ceiling in our example
- `(0 0 1 -288) ((0.0078125 0 0.5) (0 -0.0078125 -1)) "textures/hell/cbrick2b" 0 0 0`
- (0 0 1 -288)
  - vector (0, 0, 1) and the closest it reaches the origin is -288 units
  - this infinite plane will have the texture `textures/hell/cbrick2b` applied to it

Texture transformation matrix

- the texture uses the transformation matrix, T

$$T = \begin{bmatrix} 0.0078125 & 0 & 0.5 \\ 0 & -0.0078125 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

- general transformation matrix is:
- $$T = \begin{bmatrix} xscale\cos(\theta) & -yscale\sin(\theta) & translatex \\ xscale\sin(\theta) & yscale\cos(\theta) & translatey \\ 0 & 0 & 1 \end{bmatrix}$$

Each coordinate is transformed by

- $$T = \begin{bmatrix} xscale\cos(\theta) & -yscale\sin(\theta) & translatex \\ xscale\sin(\theta) & yscale\cos(\theta) & translatey \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$
- and mapped into the image file at this new grid coordinate
- fortunately we conceptualise chisel as creating a variety of lego bricks (each is a cuboid)
- `pen2map.py` generates floor bricks, wall bricks and ceiling bricks

Conclusion

- layered software is an important concept which allows large systems to be built and it can hide complexity behind well defined interfaces
- cuboids are represented by brushes in the map
  - six planes define a brush

Tutorial

- finish off your automatic light code in `txt2pen.py`
- see if you can make the floor level vary
  - by lowering slightly every odd room number floor
  - leave the even room number floor alone
- need to examine and change `pen2map.py`