

## ABSTRACT

### BRAT - (BLENDER RAPID ANIMATION TOOL)

By

Richard S. Parker

May 2013

These are exciting times in the world of computer animation. Individual artists can now produce animations which in the past required teams of artists and programmers to create, but the obstacles to an individual artist can still be daunting in terms of required manual input or technical skills. The goal of this thesis is to demonstrate a simple software tool (BRAT - Blender Rapid Animation Tool) which allows an artist to create cartoon-like 3D animations using the artist's 2D drawings, at the touch of a button. It could be useful for any artist producing 3D animations. The tool is a Python script add-on to the Blender 3D modeling system. It is a high level program which links to Blender's Python API library of low level calls to automate the manual work normally required by 3D artists to create an animation. The output of BRAT is a 3D animation file showing a character traversing a rectangular matrix of buildings.

This thesis compares the manual way of creating such an animation with the BRAT automated process. The comparison explains the benefits of using BRAT. This

thesis also discusses the design of BRAT and compares it to some other existing 3D software tools which also attempt to simplify the 3D animation production process. The conclusion discusses some future enhancements to make BRAT more useful.

BRAT – (BLENDER RAPID ANIMATION TOOL)

A THESIS

Presented to the Department of Computer Engineering and Computer Science

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

Option in Computer Science

Committee Members:

Burkhard Englert, Ph.D. (Chair)

Shui Lam, Ph.D.

Art Gittleman, Ph.D.

College Designee:

Burkhard Englert, Ph.D.

By Richard S. Parker

B.S., 1986, University of California, Irvine

May 2013

UMI Number: 1523073

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 1523073

Published by ProQuest LLC (2013). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

## TABLE OF CONTENTS

	Page
LIST OF FIGURES .....	v
CHAPTER	
1. INTRODUCTION .....	1
Background .....	1
Objective of the Thesis .....	3
Outline of the Thesis .....	4
2. SYSTEM TECHNOLOGIES .....	5
Why Python and Blender? .....	5
Python... ..	6
Blender Overview .....	11
Blender Python API .....	12
How Blender Implements Object Animation .....	15
Forward Kinematics.....	15
Inverse Kinematics.....	16
BRAT and Blender vs. Poser .....	17
Comparison of API's .....	19
A Poser Script to Create Animations .....	21
3. DESIGN OF BRAT .....	24
Development Methodology .....	24
Development Hurdles .....	25
The Manual Method of Animation Creation .....	27
How BRAT Uses Blender's Python API to Communicate with Blender ..	31
How BRAT Automates Creation of Houses and Character.....	33
How BRAT Automates Creation of Character Joints.....	37
How BRAT Automates Creation of Armature .....	40
How BRAT Automates a Walk Path for the Character .....	42

CHAPTER	Page
4. OUTPUT OF BRAT .....	46
Sample Animation Screenshots .....	46
5. CONCLUSION.....	49
Future Enhancements.....	49
APPENDIX: BRAT CODE.....	50
REFERENCES .....	88

## LIST OF FIGURES

FIGURE		Page
1.	How dynamic typing can mask static members with local members .....	10
2.	Relationship between BRAT, Blender, and Python.....	11
3.	Blender “Info Panel” and tooltips displaying UI and API equivalents.....	32
4.	The main loop for BRAT function buildHouse() .....	34
5.	The Output of BRAT function buildHouse().....	36
6.	The main loop in BRAT function copyHouses().....	37
7.	From L to R: original arm, loop cut, extruded, dup arm and shrink.....	40
8.	The main loop in BRAT function bratAnim() .....	43
9.	The main loop for BRAT function poseAnim() .....	44
10.	A set of 10 drawings with base-name "happy" for BRAT input .....	47
11.	Output number 1 of sample output animations.....	48
12.	Output number 2 of sample output animations.....	48

## CHAPTER 1

### INTRODUCTION

#### Background

The past decade has seen a quantum leap in the capabilities of 3D animation hardware and software, and thanks to advances in GPU technology and 3D modeling software, the field of animation is about to achieve, or has achieved, depending upon whose opinion one subscribes to, two historical milestones. Milestone number one is the ability to achieve video realism in a full length animation produced entirely with 3D modeling tools without importing any real world video (without “cheating”). Milestone number two is providing a 3D tool or tools which would allow a single artist to produce a feature length animation.

The first milestone, video realism, has been achieved with inorganic objects but is not quite there (subject to opinion) in terms of complex organic models such as detailed human or animal athletic movements (a precisely dancing ballerina), detailed facial expressions for example those which accompany subtle emotions or speech [1], and possibly some of the large scale panoramas (large moving crowds, rainstorms, or large numbers of small semi-independently moving objects, especially of organic or fluid nature, such as hair, water, smoke, etc.). For these difficult cases many games and animated movies sometimes rely on various doctored versions of actual video. A common complaint from gamers used to be the advertising clips for some games used



doctored video (from connecting scenes) which did not live up to the game's real time rendering engine. Adding to these challenges is the audio portion necessary for most animations, since (if we prohibit any prerecorded voice) speech synthesis also presents many challenges to the goal of life like realism. Finally, there are some gray areas to our rule of not allowing video import, since if we are using portions of photographs to texture our character or motion capture to control the character's movements, though we have not imported actual video, aren't we still cheating just a bit? Nevertheless, one has only to look at any major release of a 3D game or animation to see that the goal of video-realistic (without "cheating") 3D animation is at the very least right at our doorstep.

The second milestone, the ability for a single artist to produce a feature length animation, is more critical to individual artists perhaps, than to audiences, since audiences are pleased with any well done animation regardless of (and sometimes in direct proportion to) the amount of manpower necessary to create it. But from the viewpoint of individual artists, there are many times when they wish to have complete control over their work and until now the media which allowed this level of control (painting, novels, etc.) did not include full length animation by a single artist within a reasonable time frame unless the artist was highly skilled in 3D animation techniques.

But now with the advent of affordable animation software and hardware, we are approaching a time when any "painter", rather than painting a single still scene, can turn that scene into a world and have characters move and speak within that world, opening up the possibility of an amazing number of new animations coming from talented, but until now unknown, visual artists. And best of all these animations will be able to reflect in an undiluted way the vision of those artists. It makes an entirely new medium available to

the visual artist. How many visual artists in the past have said to themselves that they had an amazing idea for an animation but not even close to enough resources or knowledge to create it. Now, finally, the resource and knowledge barriers are beginning to fall. Cost and limitations of software and hardware are no longer barriers, but time and technical skill required of the artist still are. This is where scripts written for animation systems can be critically helpful. Scripts, especially if tailored for a particular artist or his desired animation, can at the same time save the artist laborious production time [2] and simplify some of the highly complex tasks which animations often require. This leads to achievement of the “one artist/one animation” milestone, and in a larger sense the secondary goal of this thesis is to demonstrate this milestone is now reachable.

### Objective of the Thesis

The primary objective of this thesis is development of BRAT (Blender Rapid Animation Tool), a python script running within Blender which allows a single artist to create a short animation based on his 2D drawings at the touch of a button, rather than using a time-consuming and complex manual process. This thesis compares the BRAT method with the manual method to show BRAT's advantages, and discusses the design and implementation of BRAT, and to an extent the design of Blender and the Python scripting language, since an understanding of Blender and Python is required to fully understand how BRAT works.

The animation to be created is 100 frames long. It consists of one scene, a rectangular array of one or two story buildings, textured with the user's 2D drawings, through which a simple low polygon character with armature and bendable joints, and also textured with the user's 2D drawings, walks an arbitrary path. A variety of

animations using different texture drawings or with variations in number or layout of building, can be created with BRAT, and these animations could be used for animated commercials, scenes in a game, or scenes in a longer length animation.

A secondary objective of this thesis is a comparison of BRAT and Blender to similar 3D systems which assist artists in creating animations. In a larger sense we wish to show not only a small script tool for a specific animation, but to show that the power of such tools has finally reached a level which will allow an artist to create a feature length animation, thus making the “one artist, one animation” milestone reachable to all 3D artists.

### Outline of the Thesis

Chapter 2 presents a brief introduction to the system technologies on which BRAT is based, namely the Blender 3D modeling system, the Blender Python API, and pertinent aspects of Python. It ends with a comparison of the BRAT implementation to similar 3D tools.

Chapter 3 explains the design of BRAT and how it automates the manual process of creating the animation. Some of the main algorithms and functions used by BRAT are discussed, including some code segments for illustrative purposes.

Chapter 4 demonstrates several output animations from BRAT with sample screen shots showing a typical user session.

Chapter 5 concludes with a review of the advantages of BRAT-like scripts and discusses future enhancements which would make BRAT more useful.

## CHAPTER 2

### SYSTEM TECHNOLOGIES

The software systems and content used by BRAT to create animations are:

1. Python.
2. Blender.
3. A content folder with 2D drawings in a .PNG or .JPEG format, which can be produced by any drawing program of the artist's choice, from the very simple Microsoft Paint to advanced products such as Adobe Illustrator.

#### Why Python and Blender?

A tool to create our animation could be written from the ground up, such as a C++ application linking to OpenGL and GLUT. In this case, the effort required to create the tool would be greater than the time saved once the tool automatically created animations. The alternative of using an existing 3D modeling system and creating the animation manually through the system's user interface is reasonable, but any changes to requirements for drawing textures, number of buildings, and/or walk path of the character require some repetition of the manual process.

The best solution for a fast tool is to use an existing 3D modeling system with a programmable API so that a script can be run against it which automates the manual work, since this is what API's are designed to do. Many commercially available systems come with API's, such as Maya and 3DS Max. But the price of these systems is beyond

the reach of many artists, whereas Blender is the preeminent free, open source 3D modeling system. Blender's API is written in Python, so we choose Blender as the 3D system platform for our script tool, and Python as the language to write it in.

### Python

Python is a high level, object-oriented, interpreted computer language requiring a Python interpreter to run python statements, which can be combined into scripts and run from a Python shell. The shell can be interactive and visible, but does not have to be. Python is the language used for BRAT and for the API (Application Programming Interface) into Blender 3D, including all of Blender's add-on scripts. It is also the API for many of the top 3D modeling systems commercially available such as Poser and the (currently) industry standard Maya.

Why is Python an effective 3D modeling scripting language? The goal of a scripting language is to raise the level of abstraction of the host language and to delegate some work to an external language [3]. In our case, the "host language" is actually the 3D modeling interface into Blender which can be thought of as a language if various interface commands (select a vertex, add a cube) are thought of as commands. The goal of the scripting language in this case, then, is to raise the level of abstraction of various user interface commands and/or delegate some of those user interface activities to the scripting language.

Of course, for a scripting language to raise the level of abstraction with minimal restrictions it must at least have the ability to mimic the same level of abstraction. One of the most useful features of Blender is the Blender Python API which allows access to all user-accessible (through the interface) data and functionality. In simple terms, it was

desired that any data manipulation available through the UI should allow duplication through API calls. During user sessions, this mirroring can be seen in Blender's "info" panel since for every user interface action, the "info" panel attempts to expose the equivalent Python API command which was executed to perform the same action. Since we now have a good one-to-one mapping of Python command lines to user interface actions, we can now raise the level of abstraction by combining user interface commands (translated into their Python mapped equivalent) into various scripts and/or delegate large amounts of user interface activity to those scripts.

Python is a good choice for the Blender API for several reasons. Python is flexible, partly because it is an interpreted, dynamically typed language rather than compiled, statically typed. A statically typed language binds variables at compile time to a type which the variable must remain true to at all times. Python allows a variable name to be bound at execution time to any appropriate type, and later it can be bound to a different type. This is extremely helpful both in the Interactive Console and in script development.

For example, Blender's Python Interactive Console allows the user to traverse part of the bpy module with the commands (easy to input with Autocomplete ON): `b=bpy` <CR>, `b=bpy.data` <CR>, `b=bpy.data.objects`, etc. where the type of variable `b` is changing on the fly and no recompiles are necessary. By using Autocomplete, various elements of `b` can be inspected as it “drills” down through the module, giving the user quick access to view or modify data at various levels of the module. The user can save various objects into the Python console's `__main__` namespace while traversing (for example `set: b = bpy.data.objects`), without the laborious time of writing a small

compilable program, declaring the type of variable (type(b)) within the program, and then using the program to display or save the value of b. This allows `__main__` to become a sort of workspace where structures and data can be saved, manipulated, compared, etc. on the fly. This flexibility would be unattainable if a compiled, statically typed language were the API..

In terms of script writing, consider a function whose goal is to return an addition of twice the first argument to the second argument, which can be done very quickly in Python with:

```
def foo(a,b):  
    return(a+a+b)
```

For the sake of simplicity, we will assume both argument types will always be the same. In Python, we can pass many argument types into this function, e.g. `foo("ab", "cd")` will return "ababcd", `foo(4,5)` will return 13, `foo([1,3], [8,9,10])` will return `[1,3,1,3,8,9,10]`, and more importantly for our application (if addition is defined for vertices) then `foo(vertex1, vertex2)` will return appropriately. As long as the addition method is defined for any types which will be passed into "foo" all that is needed is the above function definition, and the results are what intuitively should be returned.

We stated that we wished through scripting to abstract to a higher level various primitive operations, and to an extent abstracting assumes that "generic" commands such as addition will be interpreted as needed at the lower levels, in a manner which intuitively makes sense, which is exactly what "foo()" does. If a compiled, statically typed language is used, then overloaded methods for each argument type would have to be defined, then all the definitions compiled. A statically typed language will not allow compilation of

methods with unknown argument types, argument types have to be declared specifically. Furthermore, if the “+” operation is later defined for a new type then with Python “foo” can still be used with that new argument type with no need of function re-definition or re-compilation steps.

This flexibility extends to class definitions. A compiled language such as C++ requires a class to be fully defined before runtime. With Python a class can be defined on the fly, and then later methods and elements can be added to the class so that subsequent instances of the class reflect those new methods and elements. This is helpful for rapid script development of, for example, traversal and manipulation routines for 3D data structures where the type and layout of property variables is unknown, and thus the routines need to be adapted as quickly as possible to changing information.

We end on a cautionary note. Though legal in Python, redefinition (within the same name scope) of a variable, method, function or class is a classic syntactical error in compiled languages such as C++, so programmers sometimes almost subconsciously rely on the compiler to keep them from unwanted redefinitions, while Python offers no such safeguard. Consider the Python interactive console session (occurring in the `__main__` namespace) in the below left to right screens.



```

File Edit Format View Help
>>> class cl1():
...   statInt1 = 5
...
>>> ob1 = cl1()
>>> ob2 = cl1()
>>> ob3 = cl1()
>>> ob1
<__main__.cl1 object at 0x09DD3430>
>>> ob2
<__main__.cl1 object at 0x09DD3450>
>>> ob3
<__main__.cl1 object at 0x09DD3470>
>>> ob1.statInt1
5
>>> ob2.statInt1
5
>>> ob3.statInt1
5
>>> cl1.statInt1 = 22
>>> ob1.statInt1
22
>>> ob2.statInt1
22
>>> ob3.statInt1
22
>>> ob2.statInt1 = 9
>>> ob1.statInt1
22
>>> ob2.statInt1
9
>>> ob3.statInt1
22
>>> cl1.statInt1 = 33
>>> ob1.statInt1
33
>>> ob2.statInt1
9
>>> ob3.statInt1
33

File Edit Format View Help
>>> class cl1():
...   statInt1 = 777
...   def cl1Func(this):
...     print("this: " + str(this))
...
>>> nuob1 = cl1()
>>> nuob1.statInt1
777
>>> nuob2 = cl1()
>>> nuob1.statInt1
777
>>> type(ob1)
<class '__main__.cl1'>
>>> type(ob2)
<class '__main__.cl1'>
>>> type(ob3)
<class '__main__.cl1'>
>>> type(nuob1)
<class '__main__.cl1'>
>>> type(nuob2)
<class '__main__.cl1'>
>>> cl1.statInt1 = 88
>>> nuob1.statInt1
88
>>> nuob2.statInt1
88
>>> ob1.statInt1
33
>>> ob2.statInt1
9
>>> ob3.statInt1
33
>>> ob1.cl1Func()
Traceback (most recent call last):
  File "<blender_console>", line 1, in <module>
AttributeError: 'cl1' object has no attribute 'cl1Func'
>>> nuob1.cl1Func()
this: <__main__.cl1 object at 0x09DD7F10>

```

FIGURE 1. How dynamic typing can mask static members with local members.

At first glance, looking at the middle of the second screen, it might appear that ob1, ob2, ob3, nuob1, and nuob2 are instances of a very simple class “cl1” in the `__main__` namespace, but the “static” member `statInt1` is actually a local member of “ob2” which masks the class static member, and the new objects (nuob1, nuob2) belong to the redefined class `cl1` and thus `cl1Func()` and `cl1.statInt1` can only be invoked from them. This is due to dynamic redefinitions of these members, and this dynamic nature

can sometimes create unexpected side-effects during development.

### Blender Overview

Blender is a free, open source 3D animation program written primarily in C. It also contains an embedded Python interpreter, and comes bundled with a Python API library, and a large collection of Python scripts to facilitate or automate various tasks such as file import/export, mesh editing, etc.. These scripts can be modified by the user, or entirely new scripts can be written. The scripts can be launched directly from Blender's interface menu (if fully integrated) or from a Python shell running in one of Blender's windows. The following illustration shows the relationships between BRAT, Blender, Python scripts, and the built-in Python interpreter.

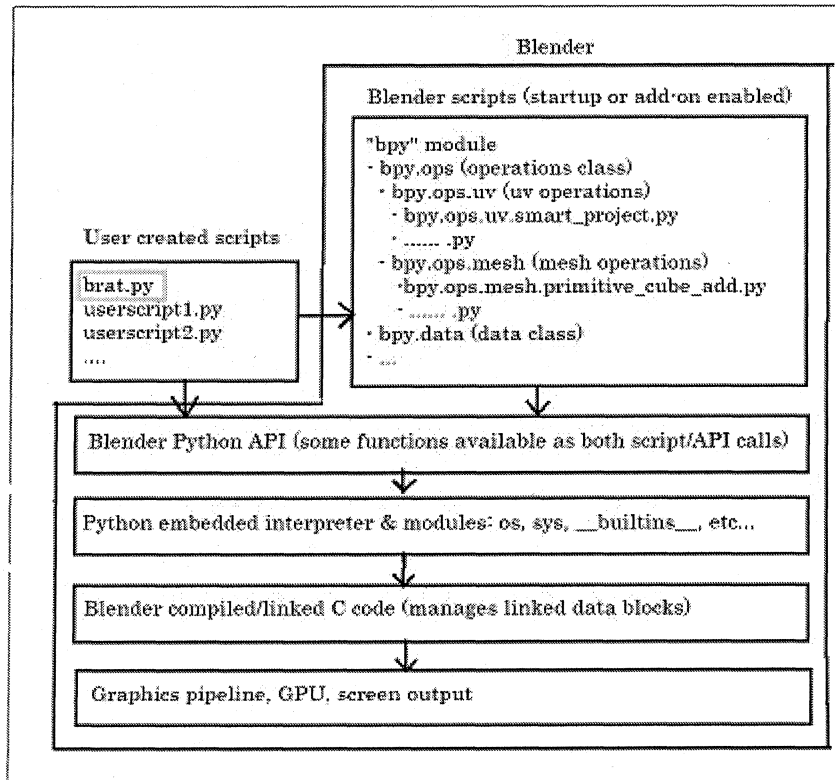


FIGURE 2. Relationship between BRAT, Blender, and Python.

So we see that BRAT, like the rest of Blender's scripts, is a Python program which calls functions defined in the Blender API library and thus is dependent upon the API to do perform lower level functions. A few typical lower level API functions are: add/delete vertex, select face of a mesh, join 2 objects, or create an armature bone.

Before we discuss BRAT, we will discuss some aspects of Python scripting design and functionality, since BRAT is simply one of numerous Python scripts which can be called from Blender's built-in Python interpreter, and makes extensive use of Blender's Python API. Just as any computer language is best understood within the context of other languages, so to is any 3D modeling system add-on, so we include in our discussions the 3D modeling program Poser Pro from Smith Micro.

### Blender Python API

To understand Blender and how BRAT functions within it, we should have a basic understanding of the Blender Python API. An Application Programming Interface (API) for any software system is a set of functions and data which allows a user-written program to communicate with that system. It is usually a library designed to be linked to from a specified language. The Blender Python API is a library of functions which a user-written Python program, such as BRAT, can link to for communication with Blender.

To process these Python programs, Blender contains an embedded Python interpreter which is started with Blender and stays active with the session. The Python shell is not returned to its initial state by closing the current *file*, to do this Blender must be *restarted*, since as mentioned previously, importing a module such as BRAT, then opening a new file from an existing session requires a repeat import AND an `imp.reload`. There is a symmetry between Python and Blender which facilitates using both, since in

Python everything is treated as an object, and in Blender one can usually assume that anything that has a name and the attributes of location and rotation will be considered an object of some sort (`bpy_types.Object`).

The Blender Python API provides a library of classes/methods which can edit any of these objects, and in fact can edit any data the user interface can (Scenes, Meshes, etc.). Almost anything that can be done by a user clicking Blender menu buttons and/or the mouse, can be done by a call to one or a more API functions. Deeper integration of any Python script into Blender is possible, such as creating new Blender menu buttons to run the script and/or extending parts of Blender with classes/methods defined in the script module, but that level of integration is beyond the scope of our project.

BRAT creates Blender objects and modifies their meshes. In a scene of 3D objects, we usually think of objects such as cubes or spheres or cones which have a shape or form defined by a mesh (`bpy_types.Mesh`) in which case the parent object is just a shell with location, rotation, and name data as in: `bpy.data.objects[name].xxx` where `xxx = [location, rotation, name]`) and a *data* member pointing to actual mesh geometry which contains vertex coordinates. Objects are listed in “`bpy.data.objects`” with their name, and Meshes are listed in “`bpy.data.meshes`” using corresponding object names. Meshes contain information such as edges and vertices, keeping in mind that “`bpy.data.meshes[name].vertices`” is pointed to by “`bpy.data.objects[name].data.vertices`” and “`bpy.data.meshes[name].edges[n].vertices`” is a 2 integer tuple (2 indexes into `bpy.data.meshes[name].vertices[...]` for the 2 end vertices of edge #n) and each “`bpy.data.meshes[name].vertices[vnum].co`” is a 3-dimensional `mathutils.Vector` for the vnum'th vertex location.

Aside from meshes, objects can be non-mesh based, such as lamps, empty objects, or cameras, which are by themselves logically formless and without meshes/vertex data. In any of these cases, name, location, and rotation still apply and `bpy.types.Object` remains the parent class and topmost entity. This fits in well with the Python API since Python also treats everything as an object, with `__builtins__.object` being the topmost hierarchy. The symmetry between Python objects and objects in the Blender world works well. A mesh object of type “cube” is instantiated not only as a geometric object in the model but also as a Python object, and changes to its properties through the API are displayed immediately in the 3D View window just as if they were done manually through the UI.

Rather than directly use Python list or dictionary types, Blender extends them with its own “`bpy_collection`” whose elements can be accessed via either index or string. This provides the advantage of being able to access large subsets of the collection through numerical loops (e.g. `bpy.data.objects[:2]` = first 2 objects in objects list) while maintaining a one to one correspondence between object names in the 3D world and in the API. However, there is no predictable mapping between names and indexes, and the index for a named object may change if, for example, new members are added to the collection. In fact, the name itself may change in cases of duplicate name collisions (assigning a duplicate name to a new object renames the originally named object), a drawback in the API since it does not yet provide the ability to define callbacks and listeners when data is changed (which would allow us to warn the user when a name collision has occurred and an object's name has been changed).

## How Blender Implements Object Animation

The visible portion of a character is simply a polygon mesh or "skin", in the case of our animation cube surfaces defined by vertices. What moves the skin, just like in a real body, is the skeleton or "armature" inside it. The walking path of a Blender character is created by combining forward motion with rotating arms and legs. Forward motion comes from moving the topmost parent bone of the armature (spine) along either the x or y axis, and limb rotation comes from rotating hip/knee and shoulder/elbow joints of the armature. The position of spine and limbs at any given moment is called the character's pose. To create the animation, the user manipulates the character into a series of poses at various frame numbers along the animation time frame, and then inserts keyframes for those poses at those frame numbers. Blender then uses a particular form of polynomial curve, known as Bezier curves, to interpolate the proper pose (values for position and joint rotation) of the character at all the remaining frames between the specified values at the keyframes. This results in a pose for every frame, which when rendered at the appropriate frames per second animates the character.

### Forward Kinematics

As described above, the character must be posed at certain intervals. Rotations must be applied to the character's arm and knee joints at selected keyframes to create a walking motion. Blender provides two methods for applying joint rotations, Forward Kinematics and Inverse Kinematics. The Forward Kinematics method follows to an extent the laws of physics, which govern the character by starting with the foot on the ground, giving the ankle a degree of forward rotation, then moving up to the knee, giving the knee some rotation, moving up to the hip, and so on, all the way back down from the

free-hanging leg to its foot, at the end of which we have all the needed information for the character's joint positions and angles.

Forward kinematics mimics this process, but is slightly more forgiving since for any given pose we can rotate any joint without necessarily following the physics order described above. But regardless of specific order, we are attempting to create an overall armature (or part of an armature) pose by rotation of individual joints which simultaneously affect any other joint positions further down the chain. This gives us excellent control over individual rotations but can make positioning the last bone in the chain or controlling the overall effect of the rotations challenging due to the chain reactions[5].

### Inverse Kinematics

Instead of trying to guess, at various keyframe/time intervals, the appropriate joint rotations from the driving foot (ankle), up the driving leg (knee, hip), and back down the free-moving leg (hip, knee, ankle) to move the character one step forward, Inverse Kinematics allows us to pivot the arms and legs in a life-like fashion by simply moving/positioning control bones. Usually these are extra non-visible bones attached to end bones such as hands and feet, or to the spine. For example, if we wish to position the leg using IK, we simply manipulate the foot or its control bone, forcing the entire leg to bend with it. For Inverse Kinematics to work, we must define IK chains which are parent child relationships through a chain of bones, and these IK chains define not only bone hierarchy but also how the bones are to interact when their control bone is moved.

To avoid ragged doll movements of the limbs, which in case of the arm would allow too many different shoulder, elbow and wrist rotations for any location of the hand,

we set up IK constraints such as the elbow joint can only rotate in a 180 degree arc, combined with stiffness factors such as the wrist joint is 2.5 times looser than the elbow joint. Using this data, Blender's IK solver decides the overall shoulder, elbow, and wrist rotations/locations based on where the hand has been moved to. In actual practice, a combination of both FK and IK is usually required to get a pose correct.

### BRAT and Blender vs. Poser

Having a basic overview of Blender, we now compare it and its associated BRAT script with some other modeling tools. As stated previously, Maya is a strong competitor to Blender, with advanced modeling features, but also operates at a level below built-in methods and content for objects such as human figures or buildings. For Maya as for Blender, these must be built from the ground up with vertices or mesh primitives.

A modeling system that does have built-in characters exists, namely Poser. Content and content management is a good starting point for comparing Blender and Poser. At one end of the spectrum, Blender is designed to give the artist maximum control over all details of a 3D scene, while at the same time leaving creation and management of content (such as buildings and figures) up to the artist. A fresh Blender install comes with some built-in textures and excellent low level special effects for particles, water, etc. but at higher levels does not (like Poser) come with a built-in library of human models including heads, hair styles, clothing items, and props. Those real world mesh constructs are at too high a level for a program which is focused on maximum control of the lower levels of animation such as vertices, edges, polygon faces, and curves. Of course, a human head or shirt is readily modeled in Blender, and in fact with very little restriction to the artist, since the artist has full control over textures, vertex



placement, etc.. And a wide community of Blender artists and freely down-loadable models exists, and non-Blender specific content can be imported into Blender through universal formats such as COLLADA, so human related models can be readily built in Blender, but nothing at this level is built in to a Blender release beforehand.

This philosophy stems from Blender's beginnings as an in-house software application for creation of commercial animations under deadline pressures in a fast paced corporate environment. The company consisted of a mix of programmers and artists, so there was no need to ease the learning curve of a fairly complex Blender UI. The UI icons, buttons, text input areas etc. are small and the menus complex to maximize input options and speed. With the end product always in mind programmers could help the artists use existing Blender features, while artists could suggest new tools for the programmers to incorporate into Blender. Full control of every detail was necessary to be able to quickly adapt to any requirements for the latest animation project, and built-in characters (such as provided by Poser) would not be helpful when each new project would require entirely new characters and no plagiarism or copyright infringement risks could be taken. Hence built-in content is minimal.

Poser was originally created by Larry Weinberg, an artist/programmer who could not find a suitable 3D modeling system for humans and so wrote his own. One could argue humans are the most interesting target of modeling, and Poser follows that path by providing an impressive built-in library of human figures (with clothes) and kinematics, and extensive tools to modify them. Poser's morph tools, thanks to a strong "human" orientation, provide one level of abstraction above Blender sculpting modes, since they can manipulate pre-defined body parts rather than general vertex groups. Morphing tools

come with proper weighting for vertices associated with human features such as nose length, forehead height, skull convexity, and jaw width. Built-in skin textures or hair or eye colors can also be applied.

Both Blender and Poser are of course orders of magnitude more sophisticated than the BRAT tool, but in a sense BRAT attempts to merge selected aspects of those 2 programs. Blender is too low-level to quickly create animations with human figures, and needs BRAT to create specific high level animation content. Poser can quickly create BRAT-like animations but only from pre-existing Poser figures which cannot easily assimilate an artist's 2D drawings and Poser cannot quickly create a layout of buildings which also use the artist's 2D drawings for textures. BRAT thus attempts to fill in a very specific niche which falls between Blender and Poser.

#### Comparison of API's

Since Blender and Poser both offer Python API's, and Poser comes with built-in content, would it make more sense to write a script for Poser than for BRAT? To answer this, we must first understand both API's. The Blender and Poser API's both allow direct manipulation of a model's mesh data, lighting, render settings, etc. through calls to a library of Python based functions, but as previously discussed Blender offers a strong one-to-one mapping of UI functions to API calls, whereas Poser offers a correlation which is not as rigorous. It is hoped Poser might follow Blender's API path, since this is a critical difference and the advantages of this design in terms of API script development cannot be overstated. To automate 3D application tasks and/or abstract them to higher levels through an API with minimum restrictions, all atomic operations must be replicable through that API. With Poser, it is not always clear what the API equivalent to

a given manual operation is.

Also, if the functions which actually modify the mesh do not care whether they are called from the UI or API, as occurs in Blender, such a design yields the ability to create chains of commands using any mixture of API calls, script tools, and UI operations, that can be stepped through forward or backward (undo), via either UI or API. This greatly enhances development of scripts, since if a particular portion of a script is malfunctioning, it can be stepped through manually while still executing the remainder of the script programmatically.

Next, after a strong mapping of UI operations to API methods, comes the issue of the API's support. For an independent artist without years of experience in a given modeling application, support (in terms of official or unofficial documentation and technical forums) is of course a major factor in successful use of the application, including its API. From a non-expert perspective, the official documentation for both Poser and Blender seemed good for the applications but somewhat lacking for the API's. Both provided API references with short descriptions for each call, but concrete examples using code snippets were unfortunately not abundant in either. Reference manuals are usually paired with usage manuals, but not in this case. Contributing factors could be that Poser is a small company and its product geared more towards artists than artist/programmers, and Blender's "official" support is a group of unpaid volunteers and the code is evolving so rapidly development takes precedence over documentation and documentation is continuously falling behind.

Since official documentation focuses on reference manuals, we are left with unofficial documentation to answer many programming questions and it is here that

Blender far surpasses Poser. A simple web based search of “Blender Python” vs. “Poser Python” demonstrates this quickly. There are numerous websites supporting Blender Python scripting, with free tutorials and source code for various scripts. And of course Blender's open source production and test scripts are freely available and can be easily modified by any user. Even if not documented, allowing a script's source to be viewed provides much insight into the API. One of the best ways to learn the API or even the entire modeling application is to make small changes to production scripts. Poser's tutorials and support sites are fewer in number, with a more pronounced marketing theme, and for many major scripts only .PYC files are provided, thus hiding the source code. This does not reduce Poser's strength in human character modeling but makes script development, such as writing a BRAT-like script, much more difficult.

#### A Poser Script to Create Animations

Despite the aforementioned development challenges to a Poser script, a useful exercise in API comparisons is discussing what would it take for a Poser script to automate some of the functions performed by BRAT in Blender, such as creating a character, texturing the character's face from 2D drawings, or making the character walk a path.

Since, as previously seen, Poser comes with a comprehensive library of human models, a BRAT-like script for Poser would not create the character from mesh primitives such as cubes or cones, and the armature from a series of bone extrusions from the spine, as BRAT does in Blender. A Poser script would first load a user-chosen character from Poser's content library, which can be done through calls to the API. But if the artist, for example, wishes to use his 2D drawings for the face or clothes of the character, the

challenge then becomes replacing the built-in Poser textures with UV mapped 2D drawings.

One option is to replace selected body parts with more flat-sided shapes so that 2D drawings can be mapped onto their surfaces. The Poser API defines a class of type Actor which most objects (body parts, cameras, lights, etc.) belong to. It exposes vertex groups for various meshes such as body parts as type Geometry. It also follows the setter()/getter() format for accessing or changing data, unlike Blender which allows direct manipulation of class members. Poser's Actor methods: Geometry() and SetGeometry() can be used to view or change the geometry of the head to a cube through a call such as: Actor["Head"].SetGeometry(Cube.Geometry()). This call unfortunately does not overlay the target geometry's origin onto the original origin, so calculations must be done to correct the new origin. Then we can UV map 2D drawings onto the cubic faces of the head as in BRAT. A similar process would be used to texture the body's clothes with 2D drawings.

To make the character walk a path, Poser comes with a library of walk styles for characters such as swagger and hip-hop, and for even more customization has an advanced "Walk Designer" to define character motion, but these are primarily available as manual tools through the UI. It was unclear based on Poser API documentation if all the steps of animating a walk path for the character could be done programmatically through API calls.

We see in a final comparison between BRAT and Poser that Poser's library of content allows creation of an animated scene with a wide choice of figures (faces, body-types, hair styles, etc.) and walk styles (swagger, skip, stagger, etc.), all of which are

much more realistic and sophisticated than similar BRAT animations. But automating through a Python script all aspects of the Poser animation (especially the character's walk path) and allowing use of the artists' 2D drawings for texturing the character are both problematic. For the very specific case of a world populated with cubic shapes and textured with 2D drawings, BRAT is a simple but effective tool and almost fully automated.

## CHAPTER 3

### DESIGN OF BRAT

#### Development Methodology

For large scale projects, various methodologies such as the waterfall method are available, but for the one developer implementation of BRAT the cowboy coding method was utilized, with some aspects of continuous integration. More specifically, the task of creating the animation was broken down into discrete steps. Each step was first implemented manually through Blender's UI, then information was gathered through documentation or from tooltips or from Blender's info panel on relevant API calls which could perform the identical step. Small scripts could then be written into the Python Interactive Console to test the API calls. A cycle of script testing and script modification was then performed until the script could perform the indicated step. The script code was then merged into what currently was the main script code for BRAT, and the test/modification cycle performed on the new version of BRAT until the new step and all previous steps functioned correctly.

The advantages of Python's dynamic typing and an interpretive language were immediately obvious during development of B.R.A.T. Some of the major obstacles encountered during development were sparse documentation, bugs in the released versions, and lack of any well-established process to run an interactive debugger on the Windows platform.

These might have posed serious barriers to script development, were it not for the ability of Python and the Blender API to run small subsets of the main BRAT script or even individual command lines quickly and repeatedly. This allowed for very fast testing of code segments to debug existing functions and prototype new ones. Functions not well documented in the API manual could be experimented with in the Python console quickly, simply by typing them in directly, with various arguments, and without the time intensive labor of compiling and linking them prior to execution.

### Development Hurdles

A slight drawback to the use of Python scripts in a fast paced development environment was the tendency for Blender's interactive Python shell to hold on to old versions of scripts, therefore forcing the need for constant “imp.reload” commands. For example, if a new interactive shell was opened, module “brat” ceased to be defined which normally would indicate a simple repetition of the “import brat” command would suffice, but in such instances an additional “reload” command was still required. This indicated a disparity between having a module not defined in a given Python shell (which is usually an indication the module has not yet been loaded) and yet having the module still loaded and listed in “sys.modules”. This is partly due to Blender's mechanism for opening a “new” Python shell, which does not entirely restart the Python interpreter. Furthermore, Python itself does not provide any simple “un-import module” process, so during fast paced script editing and testing from Blender's interactive Python console, care must be taken to ensure the latest code is being tested.

Another area which might create problems for a developer new to Python is the use of indentation to define scoping. Most languages ignore white-space and use



parentheses and brackets as scope delimiters, but Python uses indentation. Although this does tend to force code to be more readable, it can create problems when for example text information from a Blender window is imported into a Python code editor such as Cream. During development of BRAT it was found that heavy use of cutting and pasting code sometimes allowed special characters to find their way into the Cream Editor which were difficult to find or remove and which generated syntax errors by the Python interpreter. Worse, the syntax errors indicated problems with argument lists for low level API functions rather than simple high level syntax errors. These would not have been an issue with a compiler which ignored white-space, so a programmer new to Python should be prepared for them.

During development, minor inconsistencies were found in the Python API. For example, the `context.mode` property returns a different string value when displayed in the GUI than what is needed to reset the value through the `object.mode_set` command, namely "EDIT\_MESH" instead of "EDIT". Also, "`bpy.ops.object.join()`" is an API function that joins any objects which are currently selected in OBJECT mode. If the command is performed manually or entered through Blender's Python Console, and objects A and B have been selected in that order, the combined object name is B.name. Through a script calling the API `object.join()` function, the resulting joined object's name is A.name. Being able to make assumptions about the API based on the UI saves much time, so even minor deviations between the two can accumulate to slow development time. It is worthwhile remembering the guideline that "*An API should be minimal, without imposing undue inconvenience on the caller*". This guideline simply says "smaller is better." The fewer types, functions, and parameters an API uses, the easier it is

to learn, remember, and use correctly [4].

Another challenge pertaining to context effects on the API and underlying Blender model involved mesh editing. While editing an object in edit mode, an internal data structure not exposed to Python (EditMesh) becomes the target of edit changes. During BRAT development this created inconsistencies in that an object's mesh data did not reflect changes until OBJECT mode was returned to. This sometimes required repeated toggling between OBJECT and EDIT mode to verify the current mesh structure.

### The manual method of animation creation

BRAT is designed to automate the manual task of creating a simple animation. To be able to compare the automated and manual methods, we start with an explanation of the manual process. We will then cover the design of some key BRAT functions to show how they automate the manual process.

The manual method of creating the animation requires these following tasks: (1) set up the animation environment (light, camera position, number of frames, etc.); (2) create the rectangular array of buildings; (3) create the character skin; (4) create the character armature; (5) animate the character to make it walk an arbitrary path around the buildings.. We will state the operations in plain English, rather than specifying the exact sequence of Blender UI operations, with the understanding that each statement corresponds to one or a few UI operations (menu selections, text inputs, or click and drag mouse operations, etc.). For example, creating a new default size cube involves selecting [Add/Mesh/Cube] from the main menu bar.

Creating the rectangular array of buildings (Building Array operation) is done with these steps:

1. Delete all objects, since the default cube comes with default material and texture but this can lead to errors since subsequent objects do not have default textures assigned.

2. Create a new (default size etc.) cube.

3. Create new material and texture for the cube.

4. Open a uv editor window.

5. Select the front, back, and side faces of the cube

6. Specify a smart uv unwrap on the cube which "flattens" those faces into a uv layout in the editor window.

7. In the uv editor window, specify the image to use as texture ("first floor") and verify the uv map properly overlaps the image. Specify a repeat factor for the "street" texture which must be repeated (such as cobblestones or tiles).

8. In the texture properties window link the texture to the image and specify UV mapping

9. Repeat the above steps for the "second floor".

10. Repeat the above steps for the "roof".

11. Repeat the above steps (with slight variation, such as create plane rather than cube) for the "street" level.

12. Stack the 4 objects (street, first floor, second floor, roof) in the proper vertically aligned positions.

13. Combine the 4 objects into one object which gives us a 2 story building object.

14. Repeat the above steps (omitting "second floor") for the one story building.

15. Copy a mixture of one and two story buildings into a rectangular layout of users choice, by using a consistent offset to the x and y locations of each copied object.

Creating the character skin (Character Skin operation) is done with these steps:

1. Create a default cube for the head.
2. Follow steps 5 through 8 from the previous operation (Building Array operation) to add a texture (front, back, and sides of face) from a 2D drawing to the cube.
3. Repeat steps 1 and 2 from directly above five times, so that each of the 5 iterations will create one of: main body, upper arm, lower arm, upper leg, and lower leg.
4. Subdivide the head and body cubes with a single cut in each dimension and delete the left halves since we want a “vertical axis symmetry” modifier to produce the mirror image left half from the right half.
5. Position the 6 body parts (head, body, upper arm, lower arm, upper leg, lower leg) in the proper positions to create a right half-body.
6. Combine the 6 objects into one object which gives us a “right half” body.
7. Ensure the center of the half-body object is in the middle of the plane which divides the body so that symmetry works properly.
8. Apply the mirror modifier along the y-axis so the half-body is rendered as a complete body.

Creating the character armature (Character Armature operation) is done with these steps:

1. Create a new single armature bone to be the spine of the skeleton.
2. In edit mode select the “X-Axis Mirror” option for bone creation. This creates a second (mirror position) bone object for every bone created.

3. Select the appropriate end of the spine bone, and from it extrude right shoulder bone. The left shoulder bone will be created at the same time via the mirror option described above.

4. Repeat the above step to extrude upper arm from shoulder, lower arm from upper arm, etc. so that the following bones are created (along with their symmetrical counterparts which are created automatically): right upper arm, right lower arm, right hip, right upper leg, right lower leg.

5. Ensure the armature is positioned correctly relative to the character skin and make the armature a parent to the skin using automatic vertex weighing.

Animating the character (Character Animation operation) is done with these steps:

1. Take the time interval from the point where both character's feet are adjacent, with the left foot about to launch forward, to the point where both character's feet are adjacent, with the right foot about to launch forward. This is half of the “walk cycle” (the smallest sequence of movements that can be repeated indefinitely to create a walking movement).

2. Select an arbitrary (less than five) number of points along that interval and create a pose (rotate the elbow and knee joints) for each of those points so that the arms and legs are bent correctly for those points in the time interval.

3. Repeat step 1, but rather than creating the poses for the second half of the walk cycle, they can be copied from the first half and pasted in mirror image fashion.

4. Select the time intervals from steps 1 and 2 above, and combine them into an action named “walk cycle”.

5. Select an arbitrary path (series of X,Y displacements) for the character to walk

around the buildings for the length of the animation, and keyframe those X,Y displacements.

6. Keyframe the proper rotation of the character at the end of any of the displacements in step 4 which precede a change of direction (a left or right turn) so that the character remains facing forward.

7. Apply the “walk cycle” to the character, repeating it until the total elapsed time of the repetitions equals or exceeds the time expended for the character's X,Y displacements (movement around the buildings) in step 4 so that the character walks while he moves (rather than just gliding).

#### How BRAT uses Blender's Python API to Communicate with Blender

A major overhaul of Blender 2.5 architecture created a symmetry between UI and API such that any UI operation can be duplicated with the API, and in most cases it is simple to view the correlation through the “Info” panel or tooltips. For 2.5, the design specifically required that almost any task that the user could perform through a menu selection, mouse operation, keyboard input, or any other UI activity must be routed through the API or at least duplicable through the API. This aspect of Blender makes it extremely attractive to a script writer wanting to automate manual UI tasks, since for any sequence of keyboard inputs, object selections, mesh modifications, etc. that the user performs, a list of corresponding Python API calls is generated and can be incorporated into a Python script which mimics the user input.

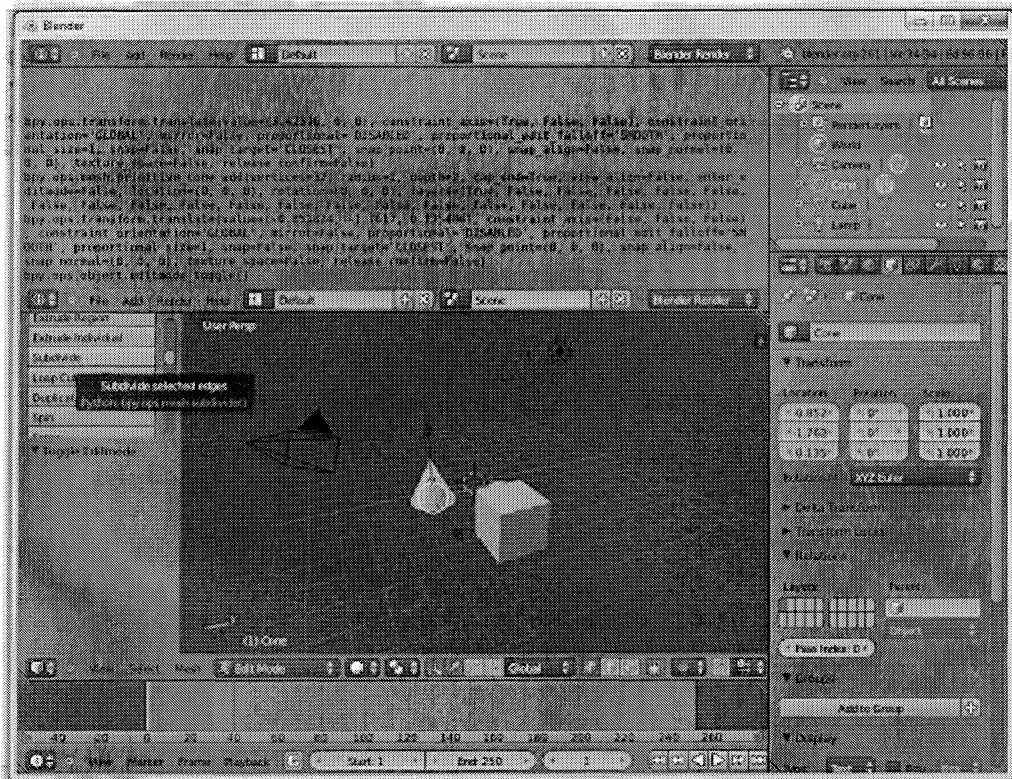


FIGURE 3. Blender "Info Panel" and tooltips displaying UI and API equivalents.

In the previous Blender console session, the user has moved the default cube, added a cone, moved the cone, and now is about to subdivide the cone through the menu selection on the left. The completed actions are visible as API calls to `bpy.ops.transform.translate()` and to `bpy.ops.mesh.primitive_cone_add()` in the info panel (top window) and the action about to be taken is visible in the tooltip as the API call “`bpy.ops.mesh.subdivide`” on the left side of the lower window.

We notice that the operations to move the cube and cone above specify only “translate” and not what is to be translated. This is dependent upon the context of the translate operation, namely what object or objects are currently selected. All operations are highly context sensitive, and context changes such as what objects or vertices are

currently selected, name changes to objects and textures, type of texture assigned to object, etc. are not fully reflected in the Info panel or tooltips. When manual operations are turned into scripts care must be taken to manage the context to avoid unexpected results. BRAT is essentially a Python script containing a mixture of calls to API functions, calls to Python built-in functions, and algorithms and data structures to provide the correct arguments to those calls and/or place them in the right context.

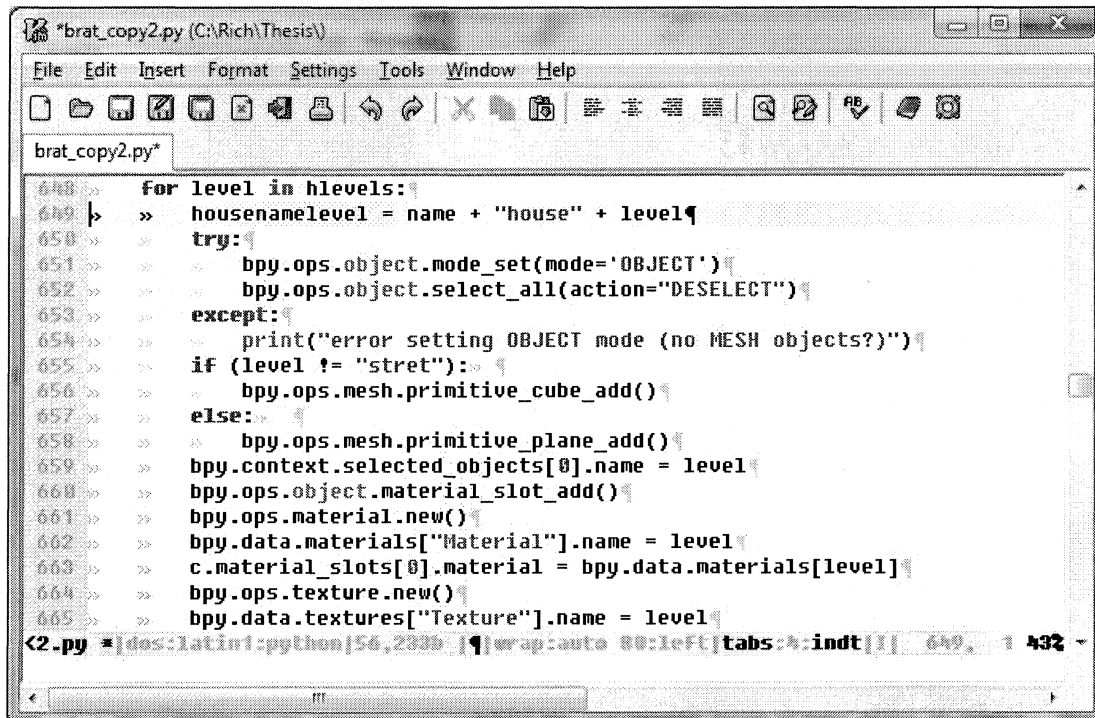
### How BRAT Automates Creation of Houses and Character

Just as with the manual method, BRAT creates buildings and a character body using 2 types of mesh objects, the plane type object (for street) and the cube type object for all body parts and all building parts other than the street. These cubes are then resized as needed, and along with the street plane joined as needed into a single object to create either a one story building, a two story building, or the character. BRAT automates this manual process by using function `buildHouse()`.

### Function `buildHouse()`

This function starts by deleting all current objects in the scene, which corresponds to step 1 in the (Building Array operation) manual process. Then it ensures the context is correct for creating cube objects. Since primitive mesh cubes are the principal building block for all the animation objects (both character and buildings), the core of `buildHouse()` is a loop which creates cubes and then performs the needed operations on them as we see in the following portion of the loop:





```
*brat_copy2.py (C:\Rich\Thesis)
File Edit Insert Format Settings Tools Window Help
brat_copy2.py*
648 >> for level in hlevels:
649 >>     >> housenamelevel = name + "house" + level
650 >>     >> try:
651 >>         >> bpy.ops.object.mode_set(mode='OBJECT')
652 >>         >> bpy.ops.object.select_all(action="DESELECT")
653 >>     >> except:
654 >>         >> print("error setting OBJECT mode (no MESH objects?)")
655 >>     >> if (level != "street"):
656 >>         >> bpy.ops.mesh.primitive_cube_add()
657 >>     >> else:
658 >>         >> bpy.ops.mesh.primitive_plane_add()
659 >>     >> bpy.context.selected_objects[0].name = level
660 >>     >> bpy.ops.object.material_slot_add()
661 >>     >> bpy.ops.material.new()
662 >>     >> bpy.data.materials["Material"].name = level
663 >>     >> c.material_slots[0].material = bpy.data.materials[level]
664 >>     >> bpy.ops.texture.new()
665 >>     >> bpy.data.textures["Texture"].name = level
<2.py *|dos:latin1:python|56,283b |wrap:auto 80:left|tabs:4:indt|1| 649, 1 432 ~
```

FIGURE 4. The main loop for BRAT function buildHouse().

The first 6 statements in the loop are to set up the correct context for creating the cubes, and we recall from above that context management is a significant task of BRAT. Then for each part named in hlevels: “street”, “face”, “upper”, “roof”, “head”, “arm1”, “arm2”, “body”, “leg1”, and “leg2” the loop creates a cube (or plane in the case of “street”) by calling the Blender API function primitive\_cube\_add() (or primitive\_plane\_add()), allowing them default arguments.

The remaining visible statements create a material and texture for each cube, and give the new material and texture names corresponding to their part, such as “arm1” or “roof”. The artist's 2D drawings need to be applied as textures to the cubes, and an object must have a material to have a texture. The remainder of the loop performs the following tasks:

1. Place the new material and texture into appropriate material and texture slots, since this is how Blender stores materials and textures. Make the cube point to that material and texture using calls to API functions `object.material_slot_add()` and `material.texture_slots.add()`. Add a repeat factor to the texture for the “street” plane. This corresponds to step 3 of the (Building Array operation) manual process.

2. Link the texture's image to the user's 2D drawing by performing a UV unwrap of the cube (or plane) onto this image using calls to API functions `uv.cube_project()` or `uv.smart_project()` as appropriate. Prior to the unwrap, the proper faces on the cube must be selected using BRAT function `selectSideFaces()` which does this by inspecting the face index list stored in API list `object.data.face[]`. This corresponds to steps 4-8 of the (Building Array operation) manual process.

3. Size the cubes appropriately using the sizes dictionary type contained in the constants section of BRAT. For the “head” and “body” cubes, subdivide the cubes and delete the left halves by calls to the API functions `mesh.subdivide()` and `mesh.delete()`. Determining the “left half” faces requires inspection of their center positions from API property `object.data.faces[].center`. For each of the upper arm and leg cubes, create 4 joints (4 extra cubes) by calls to API functions `object.duplicate_move()` and `ops.transform.resize`. This corresponds to steps 4-5 of the (Character Skin operation) manual process. We give a more detailed description of this process below.

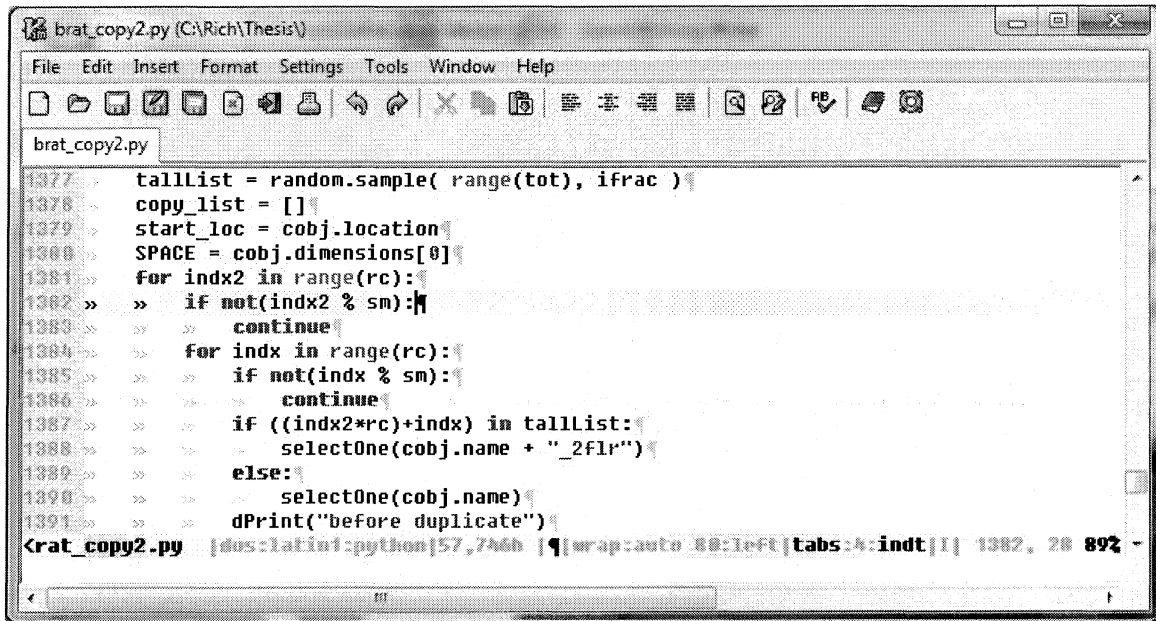
4. Once the above loop has completed, BRAT function `combineBody()` is called to join the individual house/body parts into 3 objects: a one story house object, a 2 story house object, and a character body object with bendable joints. `combineBody()` selects the appropriate cubes and calls the API function `object.join()`. This corresponds to steps

13 (Building Array operation) and step 6 (Character Skin operation) of the manual process. At the end of the call to function buildHouse() a 1-story building, 2-story building, and a character have been created as displayed below. X-axis symmetry for the character has been manually turned off so we can see the mesh actually built for it.



FIGURE 5. The Output of BRAT function buildHouse().

The final step of copying the buildings into a rectangular layout is performed by BRAT function copyHouses(). A portion of the main loop for copyHouses() is shown below.



```
brat_copy2.py (C:\Rich\Thesis\
File Edit Insert Format Settings Tools Window Help
brat_copy2.py
1377 > tallList = random.sample( range(tot), ifrac )
1378 > copy_list = []
1379 > start_loc = cobj.location
1380 > SPACE = cobj.dimensions[0]
1381 > for indx2 in range(rc):
1382 >> if not(indx2 % sm):
1383 >>> continue
1384 >>> for indx in range(rc):
1385 >>>> if not(indx % sm):
1386 >>>>> continue
1387 >>>>> if ((indx2*rc)+indx) in tallList:
1388 >>>>>> selectOne(cobj.name + "_2flr")
1389 >>>>>> else:
1390 >>>>>>> selectOne(cobj.name)
1391 >>>>>>>> dPrint("before duplicate")
<brat_copy2.py |dos:latin1:python|57,746b |wrap:auto 88:left|tabs:4:indt|| 1382, 28 892 -
```

FIGURE 6. The main loop in BRAT function copyHouses().

The 2 nested for loops control the X and Y positioning of each building. In addition, we require that a user specified fraction of the buildings be two story, and randomly placed in the grid. To do this we see in the first statement that a random sampling ( $\text{ifrac} = \text{percentage of 2 story buildings}$ ) of the total range of buildings ( $\text{range}(\text{tot}) = [1,2,3,\dots,\text{tot}]$ ) is placed in the list “tallList”. The inner for loop determines if the index number for a 2 story building has been reached (if the number appears in tallList), and selects either a 1-story or 2-story building accordingly. Subsequent statements use  $\text{indx}$  and  $\text{indx2}$  values to copy the selected building to its place on the grid.

### How BRAT Automates Creation of Character Joints

As described above, the final task for the main loop of function buildHouse() is to create joints for the character. The rationale is that, although highly cubic in shape, we

wish our character to have some realistic bending at the joints, and avoid for example the sharp outer edge of an elbow created from joining two cubes at a right angle. A simplistic approach would be to add an object display modifier such as “smoothing”, but smoothing does not work well when the polygon count is low as in our case.

The alternatives are using some form of curve based geometry to fill in the space that the joints would normally occupy, but this is too complex for our purposes and makes the mesh for the character no longer homogenous. The only remaining choice is to increase the number of vertices (and therefore polygons) at the joint areas so that Blender's automatic vertex weighing algorithm has a larger sampling of points to work with. The three most straightforward ways to add polygons at the joint areas would be:

1. Partition the 2 ends of the 2 cubes adjacent to the joint space using either "loop cut", or more problematically a "subdivide", since loop cut is more straightforward for splitting up the ends of a cube rather than the entire cube.
2. Extrude a series of thin cubes from one or both ends of the cubes adjacent to the joint space.
3. Create duplicate cubes, using their existing UV maps, shrink them along Z axis, insert into the joint space, and merge vertices with limb.

Method 3 was chosen because it yielded the least offensive artifacts (some artifacts will be inevitable if we take the shortcuts of avoiding new UV mapping/images for joint cubes). Method 3 would shrink the texture along the z-axis, but as long as we use a pattern which runs parallel to the z-axis such as pinstripping the effect is minimal. Method 3 also allows later addition of an image/UV map combo for joint cubes.

Unlike Method 3, both the loop cut and extrusion methods created problems with

the UV mapped textures because even simple changes to the mesh can invalidate the original UV mapping. A UV mapping is a one to one mapping of 3D vertices onto a 2D (UV) plane and can be confused by subsequent addition of 3D vertices. In the case of the loop cut, the UV mapping does not assume a default approach for vertex additions by preserving the map of the original n 3D vertices onto the original n 2D vertices and then dealing with the new vertices. Even if the loop cut is positioned near the end of an elongated cube, in the UI a loop cut starts in the middle of the cube, bisecting the current UV map, and as the cut is dragged toward an end it drags half the UV mapping along with it, so that half the texture now covers most of the cube, and the other half is squeezed into a small fraction of the cube (see second arm below). This behavior is replicated with an API based loop cut, and is counter-intuitive, the image should be mapped onto the entire (loop-cut) face after the cut is dragged, thereby not altering each half. The following screen shot displays the results of the different methods. The rightmost arm, created with method 3, has the best joint and no deformity of the hand.

BRAT automates the process of copying upper arm and leg cubes, shrinking them, then combining appropriate vertices, through a loop in `buildHouse()` which for 3 iterations copies the upper arm or leg cube, shrinks it, displaces it downward along the Z axis, and then merges it's vertices with the previous cube.

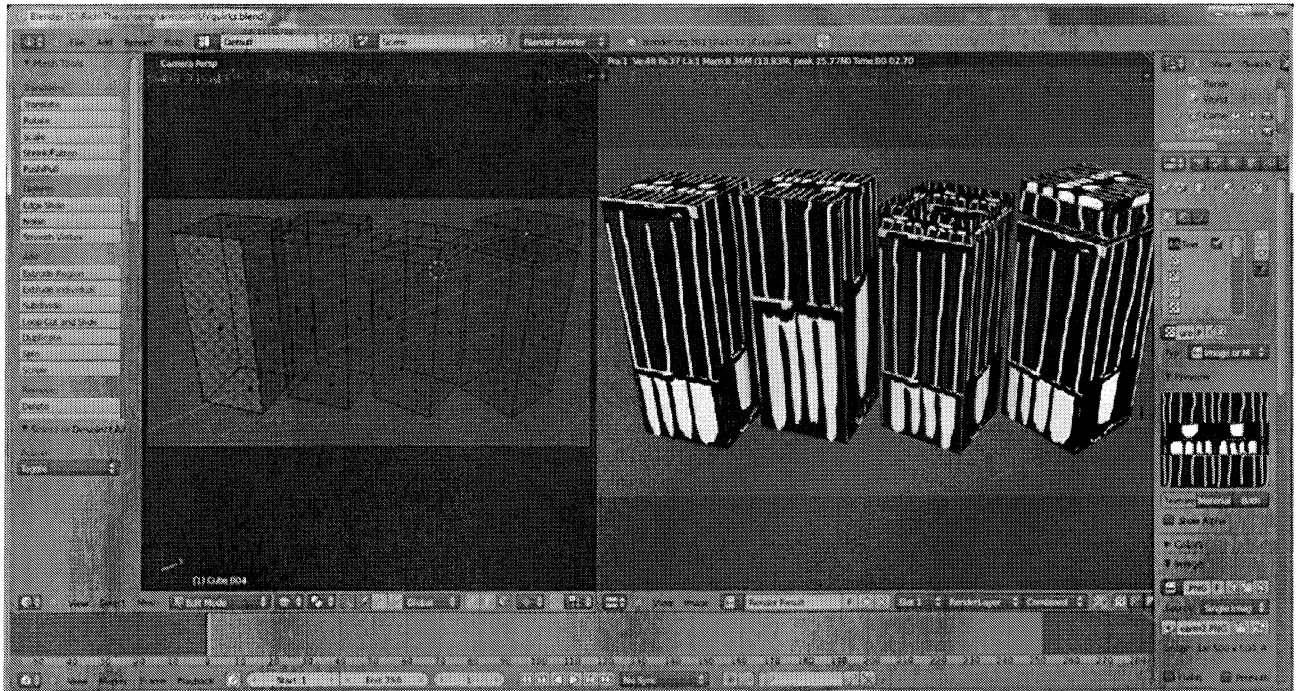


FIGURE 7. From L to R: original arm, loop cut, extruded, dup arm and shrink.

### How BRAT Automates Creation of Armature

An armature is required to animate the body. The armature bones are accessed through the API object list: `bpy.data.armatures[“Armature”].bones[...bone_name...]`. Initially, one bone (spine) is created, positioned vertically with the head at the bottom. The rest of the bones are extruded from either the head (for hip and legs) or tail (for shoulders, arms, and skull) of the spine. Normally, only the bones on one side of the armature would need to be created, and by using the mirror modifier on the armature and mirror extrusion mode, bones for the other half would be created at the same time as their counterparts. Although Blender's API provides functions and properties to implement mirroring of armature bones, during BRAT development it became evident armature mirroring functioned only during manual creation through the UI but not when used

programmatically.

The interface for building the character's armature allows for specifying X-Axis mirror for the armature while in EDIT mode. This automatically creates and names a Bone\_L... child bone for every bone extruded from a Bone\_R... sequence. Unfortunately, although all context variables (such as Armature.use\_mirror\_x) and the bone objects were available through the API, it was not possible to programmatically mimic the building of an armature in this fashion. Due to a flaw in the API, the “armature.extrude\_forked” API function could not create bones in the correct position in the bone chains without manual input from the user to manually change the position of the latest bone extruded. Only a minimal change to the latest extruded bone in the chain was sufficient, but absolutely required. Therefore, the buildArmature() function had to “manually” create and name both the left and right sequence of bones for the character's armature. Although adding time to the development effort, and complexity to the design, it made a user sensitive portion of the script more robust and pointed out one of the challenges and/or risks of creating an API to mimic user input, namely sometimes there are implicit state variables to the user interactive session which are difficult or impossible to replicate through the API. In this case, the “armature.extrude\_forked” function is apparently sensitive to side effects from cursor movement and/or other interactive session state variables beyond the arguments passed to that function. Although debugging this flaw was beyond the scope of this project, one could conjecture that the chain of functions and arguments called from the time the user begins dragging a newly extruded bone from the tail of the parent, to the time the user releases the mouse button to peg the ending position of the extruded bone at the given cursor position, for some reason allowed the final call to



work properly. A single call to extrude to a given final position with no user interaction is not the same, since it has not been preceded by a long series of previous calls to create and/or elongate and/or draw/refresh the manually “dragged” bone being stretched to its final position.

For example, the UI created bone begins with (almost) no length, since it is dragged out from its base position on the parent bone, while the API bone is created with full length. A rigorous one to one mapping of UI activity to API functions would therefore require the `armature.extrude_forked` function to be passed cursor motion data. Though removal of intermediate cursor data would seem to be more logical, much simpler, and decrease the chance of design flaws, in this case it masked a design flaw instead.

#### How BRAT Automates a Walk Path for the Character

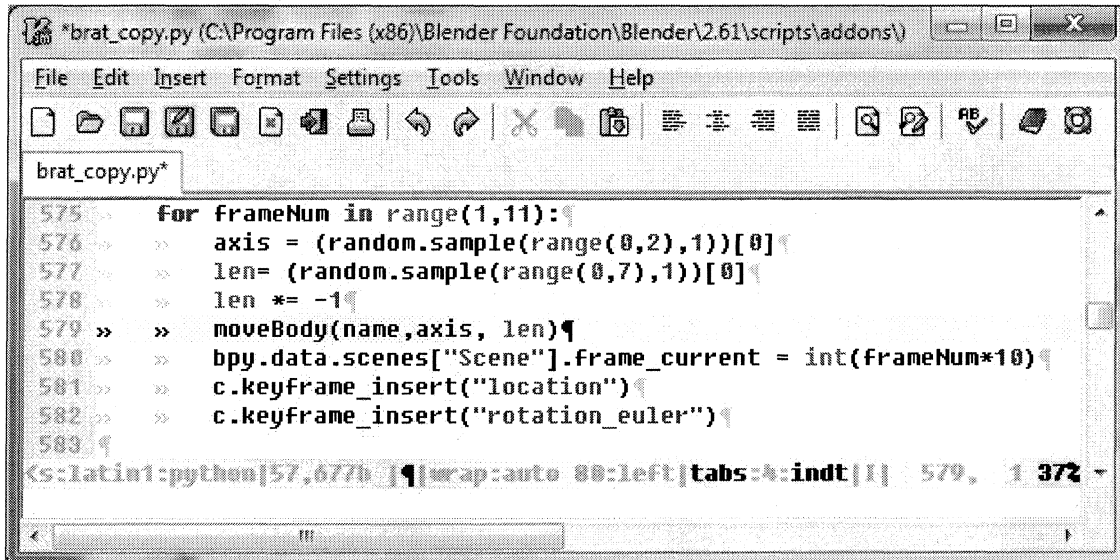
The character animation consists of 3 layers. The armature location and rotation layers control where the character is positioned in the building grid, and what direction he points in. These correspond to step 4 (Character Animation operation) of the manual method. The bone rotation layer controls how the characters arms and legs bend and straighten again as he travels forward. It corresponds to steps 1 through 3 and 5 through 6 of the (Character Animation operation) manual method. Combining the 3 layers produces a character walking a path around the buildings.

#### Function `bratAnim()`

BRAT automates the armature location and rotation layers principally using function `bratAnim()`. This function creates 10 equi-distant keyframes along the full 100 frame animation (frame 10, 20, ..., 100) and using a random function specifies at each of

these frames whether the character should go straight or turn left, and for what distance.

The main loop in this function is shown in the following figure.



```
*brat_copy.py (C:\Program Files (x86)\Blender Foundation\Blender\2.61\scripts\addons\)  
File Edit Insert Format Settings Tools Window Help  
brat_copy.py*  
575 >> for frameNum in range(1,11):  
576 >> >> axis = (random.sample(range(0,2),1))[0]  
577 >> >> len= (random.sample(range(0,7),1))[0]  
578 >> >> len *= -1  
579 >> >> moveBody(name,axis, len)  
580 >> >> bpy.data.scenes["Scene"].frame_current = int(frameNum*10)  
581 >> >> c.keyframe_insert("location")  
582 >> >> c.keyframe_insert("rotation_euler")  
583 <<  
<S:latin1:python|57,677b| |wrap:auto 88:left|tabs:4:indt| | 579, 1 372 -
```

FIGURE 8. The main loop in BRAT function bratAnim().

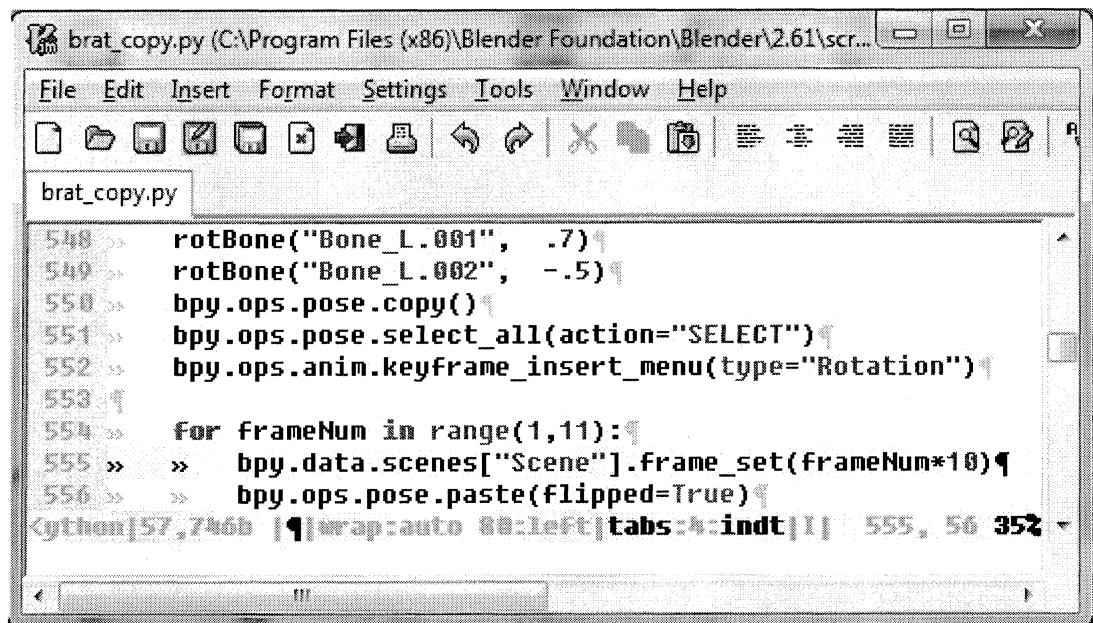
We see from the preceding code segment that Python's random.sample() function is used to generate a random choice of movement along either of 2 axis (X or Y axis) and a random distance from 0 to 6. Since we constrain movement from top left to bottom right of the buildings grid, we multiply displacement by -1 to ensure movement is left along X or down along Y. The keyframe\_insert() function is a Blender API function applied to object "c" which is the armature. We see also that object properties are exposed not only through class functions but properties can be accessed directly without need for getters or setters (frame\_current = ...). This illustrates a typical code segment for BRAT or any other Blender script which usually contains a mixture of Python module calls, API function calls, and direct manipulation of object data or scene data exposed

through Blender object lists (bpy.data) such as bpy.data.scenes[...].

Once direction and displacement have been determined, BRAT function `moveBody(name, axis, len)` is called to rotate the object if its direction has changed and to move it the appropriate distance by calling the API function `bpy.ops.transform.translate()`. Completion of the loop is equivalent to completing step D4 of the manual method.

#### Function poseAnim()

The bone rotation layer is created principally by BRAT function `poseAnim()`. A portion of the code including the main loop is shown below.

A screenshot of a Python script editor window titled "brat\_copy.py (C:\Program Files (x86)\Blender Foundation\Blender\2.61\scr...". The window has a menu bar with "File", "Edit", "Insert", "Format", "Settings", "Tools", "Window", and "Help". Below the menu bar is a toolbar with various icons. The main area of the window shows a Python script with the following code:

```
548 >> rotBone("Bone_L.001", .7)
549 >> rotBone("Bone_L.002", -.5)
550 >> bpy.ops.pose.copy()
551 >> bpy.ops.pose.select_all(action="SELECT")
552 >> bpy.ops.anim.keyframe_insert_menu(type="Rotation")
553
554 >> for frameNum in range(1,11):
555 >> >> bpy.data.scenes["Scene"].frame_set(frameNum*10)
556 >> >> bpy.ops.pose.paste(flipped=True)
<ython|57,746b |wrap:auto 80:left|tabs:4:indt|1| 555, 56 352
```

FIGURE 9. The main loop in BRAT function `poseAnim()`.

The function `poseAnim()` sets the mode to "POSE" which is required for posing the bones, and sets the animation frame to 1, the beginning of the animation. It then

rotates the bones in the upper arms, lower arms, upper legs, and lower legs to match a pose of one-half step forward with the left foot. This corresponds to step 1 of the (Character Animation operation) manual method. It then loops through the 100 frame animation at 10 frame intervals and alternately copies this pose, or it's mirror (flipped) version, onto the character and inserts a keyframe for that pose at the given frame number. This corresponds to steps 2 and 6 of the (Character Animation operation) manual method. No “walk cycle” (step 3) is created, since the automation provided by the for loop makes it easier to simply copy individual poses along the entire character's path.

## CHAPTER 4

### OUTPUT OF BRAT

#### Sample Animation Screenshots

Creating animations from BRAT requires only a few words of text input from the user. Once Blender is launched, the user opens a Python console window and imports the BRAT module. The BRAT function `run_auto()` is then called, and the animation is created. Some typical animations are shown in the following screenshots. Variations in the animation are defined by changing the default input arguments such as drawing base-name. Function `brat.runArgs()` displays an arguments summary. There are ten 2D drawings required for texturing. They should all start with a common base-name, such as “happy” (which is the default name), followed by the standard suffixes shown below.

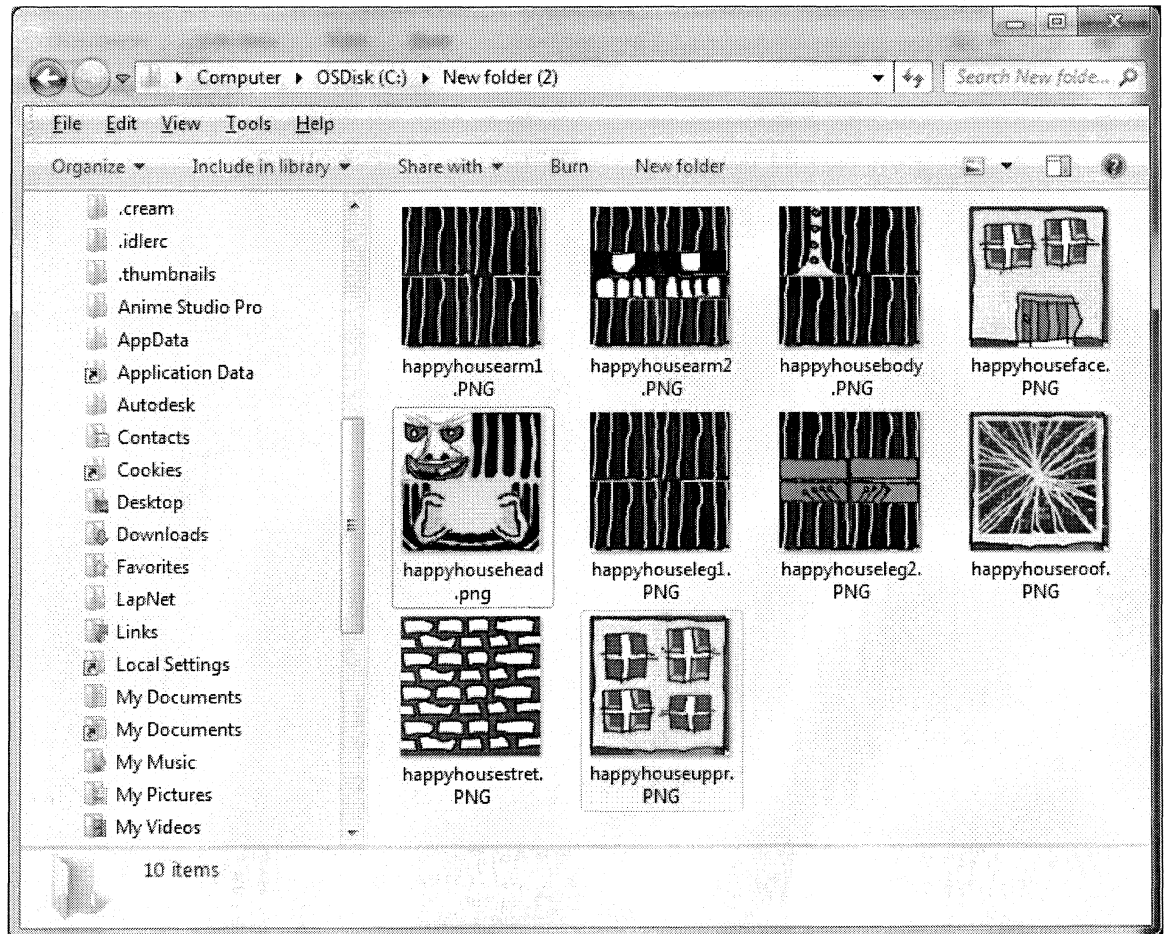


FIGURE 10. A set of 10 drawings with base-name "happy" for BRAT input.

In the previous set of drawings, we have .PNG files named: “happyhousearm1”, “happyhousearm2”, “happyhousebody”, etc.. If we wish to use base-name “crazy”, then the names would be: “crazyhousearm1”, “crazyhousearm2”, etc..

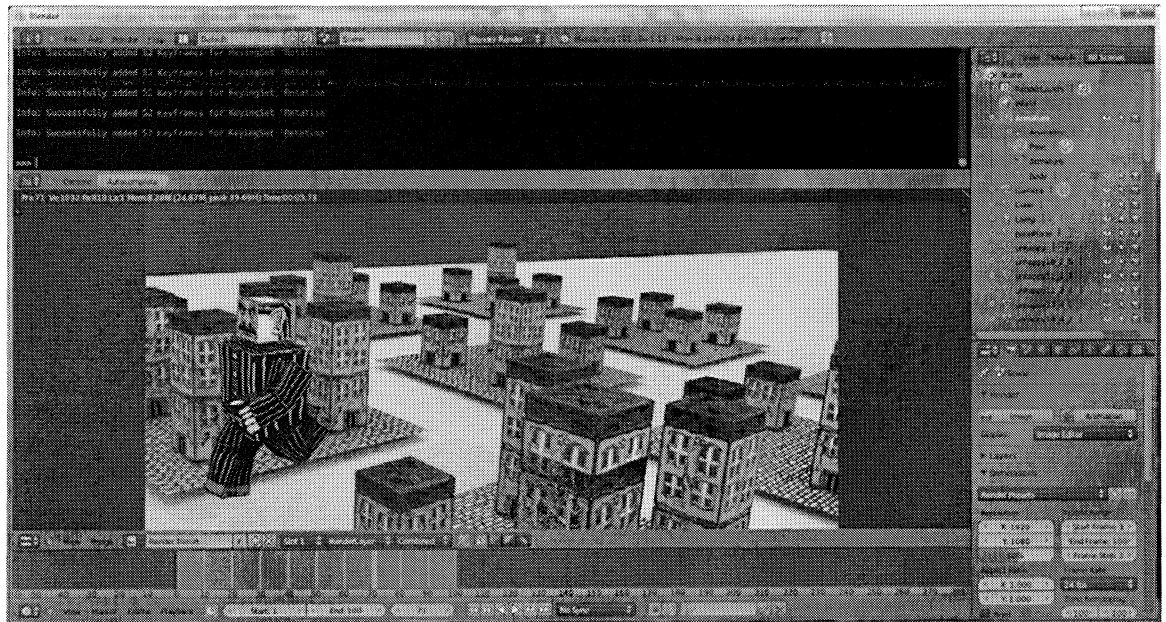


FIGURE 51. Output number 1 of sample output animations.

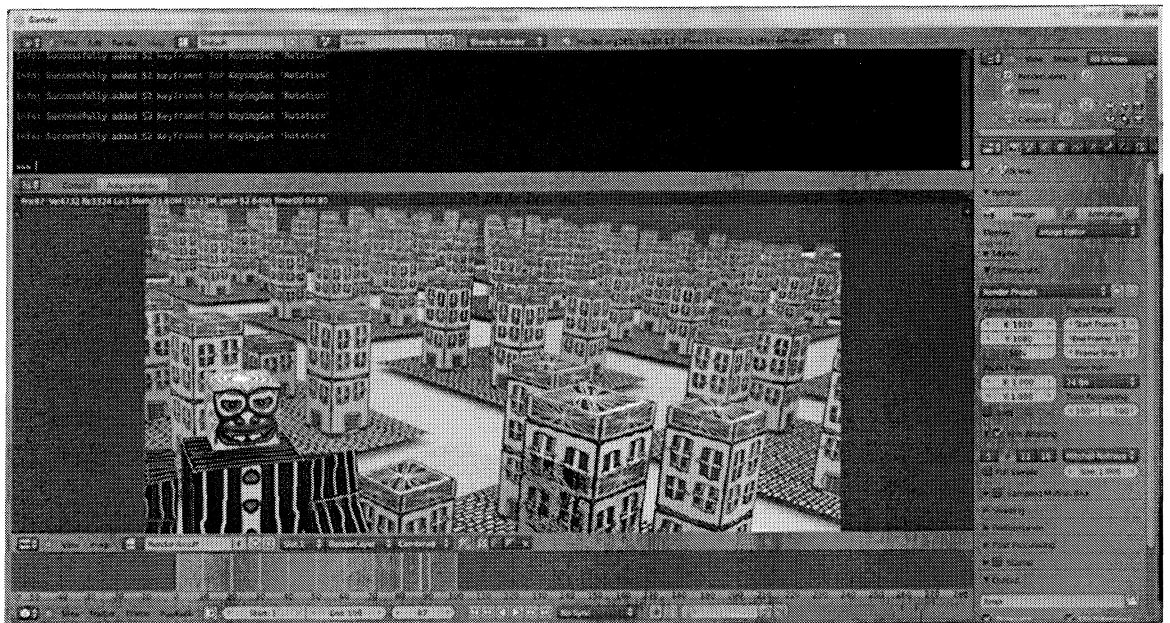


FIGURE 62. Output number 2 of sample output animations.

## CHAPTER 5

### CONCLUSION

A Python script named BRAT was written for the open source Blender 3D modeling application. The goal of being able to use the script to quickly create short 3D animations from an artist's 2D drawings was successfully achieved. As seen in the screenshots of created animations, in a matter of minutes, any artist with no programming knowledge could use BRAT to create a number animations where textures for the character and buildings textures are mapped from an assortment of the artist's 2D drawings, and various building layouts and character walk paths can be chosen from.

#### Future Enhancements

Future enhancements to BRAT might include:

1. Use Python random functions to make buildings and their layout more asymmetrical.
2. Add ability to have more than 1 character traversing the buildings.
3. Develop a "script for the script", namely a configuration file which could be parsed to create varied layouts of buildings, and control where the characters walked.



APPENDIX

BRAT CODE

## PYTHON CODE OF BRAT

```
#####
#   B.R.A.T.  Blender Rapid Animation Tool
#   To run, type "import brat" then "brat.run_auto()" from Blender Python console
#   Look at run_auto() function to see list of optional (sub-functions) arguments
#####

import bpy
import os
import re
import mathutils
import random
import math
import time
import sys

##### for debugging output #####

#####
def dPrint(text):
#####
    tstr = str(time.strftime("%H:%M:%S"))
    dlev = debugLevel

    if (dlev["dlev"] != 0):
        print("dPrint dlev...debug console continue ")
        print(str(dlev["debug"]))
        print(str(dlev["console"]))
        print(str(dlev["continue"]))
    if (dlev["debug"] == 0):
        return
    if (dlev["continue"] != 0):
        junk = input("enter any to continue:")
        print("you entered " + junk)
#   bug on certain systems when selectOne/dPrint "uppr"
    if (dlev["console"] != 0):
        print(text)
        dbg_file = open("C:\\Blender\\debug.txt", "a")
        dbg_file.write("\n" + tstr + ": " + text)
        dbg_file.close()

#####
```

```

def two_print(msg):
#*****
# for some reason bug with putting print() inside dPrint()
    print(msg)
    dPrint(msg)
#*****
***

debugLevel = {"dlev":0, "debug":1, "console":0, "continue":0, "matexs":1,
"selObjects":1}
dl = debugLevel

cmdLineText = "arguments: "
for i in range(len(sys.argv)):
    cmdLineText = cmdLineText + "<" + sys.argv[i] + ">"
two_print(cmdLineText)

#***** some constants that won't frequently change, yet *****

BODY_ROT = mathutils.Euler((0.0, -0.0, -1.57), "XYZ")
PATH_START = mathutils.Vector((0, 0, 0))
SET_CAMERA_POSITION_MANUALLY = True
#armature to move body part joints
BUILD_ARMATURE = True
#combine all the body parts into one
#required if we want to animate with armature
#joints still move but no longer addressable by name
COMBINE_BODY = True
#path for character (armature is parent) to follow
CREATE_PATH = False
MESH_SELECT_MODE = "nochange"
#***** end of switches *****

# ***** rendering settings *****

bpy.data.worlds["World"].light_settings.use_environment_light = True

newWindows = False
cobblestone_x_rep = 3
cobblestone_y_rep = 3

```

```

#folder/directory to contain 2D drawings used by BRAT
paths = {"drawings":"C:\\Blender\\"}

#---- standard house or body parts (drawings) list
# ` a level is 4 letters long
#house/body levels
hlevels = [ "stret", "face", "uppr", "roof",
            "head", "arm1", "arm2", "body",
            "leg1", "leg2" ]

bodyPartNames = ["head", "arm1", "arm2", "body", "leg1", "leg2"]

sizes = { "stret":[1, 1, 1], "face":[ 1, 1, 1.25], "uppr":[ 1, 1, 1.25], "roof":[ 1, 1, .25],
          "head":[.6,.6,.6], "arm1":[.5, .5, .5], "arm2":[.5, .5, .6],
          "body":[1,1,1.25],
          "leg1":[.5, .5, .5], "leg2":[.5, .5, .6] }

#initial positions important for combining into single objects

#body positions
#anchor everything around body
positions = {"body":[0, 0, 0]}
for part in bodyPartNames:
    positions[part] = [0,0,0]

#space in x and z directions to leave between adjoining faces
zspace = sizes["body"][2]/10
xspace = sizes["body"][1]/10
#KLUGES... maybe easier to use translate once part is created???
positions["head"][2] = positions["body"][2] + (sizes["body"][2])/2 + sizes["head"][2]/2 +
zspace
positions["leg1"][1] = positions["body"][1] + (sizes["body"][1])/3
positions["leg1"][2] = positions["body"][2] - (sizes["body"][2])/2 - sizes["leg1"][2]/2 -
zspace
positions["leg2"][1] = positions["leg1"][1]
positions["leg2"][2] = positions["leg1"][2] - (sizes["leg1"][2])/2 - sizes["leg2"][2]/2 -
zspace
positions["arm1"][1] = positions["body"][1] + (sizes["body"][1])/2 + sizes["arm1"][1]/2
+ xspace
positions["arm1"][2] = positions["body"][2] + (sizes["body"][2]) - sizes["arm1"][2]/2 -
zspace
positions["arm2"][1] = positions["arm1"][1]
positions["arm2"][2] = positions["arm1"][2] - sizes["arm1"][2]/2 - sizes["arm2"][2]/2 -
zspace

```

```

# ***** DEFINE STRUCTURES *****

# screenplay input line "targets"
# i.e. allowable regular expressions for input lines (patterns)
#class scplay_exp():
#    _fields_ = [("pat", str * 2)]

# what status is parser in (string values)
# status values (combin/correlate with above ASAP)
#class stat_strng():
#    _fields_ = [("stat", str * 2)]

#***** INITIAL REG EXP VALUES FOR FUTURE ANIMATION
LANGUAGE *****

scplay_exp = ["dummy", "(INT.|EXT.)([a-z]+)(DAY|NIGHT)", "(runto.|runfrom.)([a-
z]+)(STRAIGHT|ZIGZAG)"]
stat_strng = ["dummy", "readin scene header", "readin action1"]

#*****
# this section contains small utility functions to aid development
#*****

class testClass:
    testInt = 8888
    def cfun(a, b):
        print("first arg..")
        print(a)
        print("second arg...")
        print(b)

def listt(iter):
    if iter:
        for x in iter:
            print(x)
    else:
        pass

def slist(alist):
    print(str(list(alist)))

#def listMeshVerts():
def lmv():
    for m in bpy.data.meshes:

```

```

print("===== mesh for object named: " + m.name)
print("----- VERTICES-----")
for v in m.vertices:
    print("    ---- vertex---- " + str(v.index))
    print(v.co)
print("----- EDGES-----")
for e in m.edges:
    print("    ---- edge ---- " + str(e.index))
    print("    list vertices (no coords for edge verts)")
    for v in e.vertices:
        print("            " + str(v))

def listBones():
    print("bases.....")
    listt(bpy.context.selected_bases)
    print("bones.....")
    listt(bpy.context.selected_bones)
    print("objects.....")
    listt(bpy.context.selected_objects)
    print("editable bones.....")
    listt(bpy.context.selected_editable_bones)
    print("pose bones.....")
    listt(bpy.context.selected_pose_bones)
    print("sequences .....")
    listt(bpy.context.selected_sequences)
    print("editable sequences bones.....")
    listt(bpy.context.selected_editable_sequences)

def updateObject():
#    in EDIT mode, changes to OBJECT are not permanent until switch back to
OBJECT mode
#    this can yield unexpected values returned from object inspection methods
saveMode = bpy.context.mode
bpy.ops.object.mode_set(mode='OBJECT')
fixModeSet(saveMode)

#set current mode to POSE mode
def setModePose():
    if bpy.context.mode == "POSE":
        pass
    else:
        bpy.ops.object.posemode_toggle()

# select one face from a mesh object

```

```

# where index is index number (integer) of face to be selected
def selFace(n):
    ob = bpy.context.active_object
    for f in ob.data.faces:
        f.select = False
    ob.data.faces[int(n)].select = True

def testregex():
    c = "some stuff"
    while (c != "exit"):
        newpat = (input("enter pattern: "))
        if (len(newpat) > 2):
            pat = newpat
            print("pat = ****" + pat + "****")
            c = (input("enter sample line: "))
            print("findit = re.match(" + pat + "," + c + ")")
            findit = re.match(pat, c)
            if findit:
                print("groups(): ")
                for g in findit.groups(0):
                    print("<" + g + ">")

def renderit():
    bpy.ops.render.render(animation=True)

def abc():
    print("abc")

#*****
def matexs(message):
#*****
    if (debugLevel["matexs"] != 1):
        return
    print("matexs: " + message)
    print("-"*40)
    print("bpy.data.objects")
    for i in bpy.data.objects:
        print(i)
    print(".....selected objects...")
    for o in bpy.context.selected_objects:
        print(o.name)
    print(".....end of selected objected....")
    print("-"*40)

```

```

print("bpy.data.images")
for i in bpy.data.images:
    print(i)
print("-"*40)
print("bpy.data.materials")
for i in bpy.data.materials:
    print(i)
print("-"*40)
print("bpy.data.textures")
for i in bpy.data.textures:
    print(i)
print("-"*40)
print("bpy.data.meshes")
for i in bpy.data.meshes:
    print("-----")
    print(i)
    print("for this mesh: uv_textures = ...")
    for jj in i.uv_textures:
        print(jj)
    print("for this mesh: uv_textures.active = ...")
    print(i.uv_textures.active)
print("-"*40)
print("***** SLOTS for all MESH OBJECTS *****")
for c in bpy.data.objects:
    if (c.type != "MESH"):
        print(c.name + "not a MESH, continuing...")
        continue
    print("----obj name: " + c.name)
    print("material slots for <" + c.name + "> are below...")
    for ms in c.material_slots:
        try:
            print(ms)
            print(" ---- textures for this mat ---")
            try:
                for j in ms.material.texture_slots:
                    print(j)
            except:
                print("error printing texture slot")
                break
        except:
            print("error printing material slot")
            break

    try:
        print("intensity for first obj/mat/text")

```



```

        print(c.material_slots[0].material.texture_slots[0].texture.intensity)
    except:
        print("cannot print intensity")

if (dl["continue"] == 1):
    usercont = input("end of matexs <" + message + "> enter c to continue: ")

def listObjMods():
    for obs in bpy.data.objects:
        print("***** " + obs.name + " *** modifiers *****")
        for m in obs.modifiers:
            m

def selectOne(obj_name):
    dPrint("obj_name = ")
    dPrint(obj_name)
    dPrint("About to select objects[" + obj_name + "]")
    try:
        print("active: " + bpy.context.active_object.name)
    except:
        print("in selectOne, no active object, try to set it...")
        try:
            bpy.context.scene.objects.active = bpy.data.objects[obj_name]
            print("active set to: " + bpy.context.active_object.name)
        except:
            print("in selectOne, no active object and cannot set")
    dPrint("selected: ")
    selObjects()
    try:
        bpy.ops.object.mode_set(mode='OBJECT')
        bpy.ops.object.select_all(action="DESELECT")
        bpy.ops.object.select_name(name=obj_name, extend=False)
    except:
        print("Error selecting objects[" + obj_name + "]")
        print("-- verify: change mode to OBJECT and/or object exists")
    dPrint("DONE WITH select objects[" + obj_name + "]")
    try:
        dPrint("active: " + bpy.context.active_object.name)
    except:
        dPrint("in selectOne, no active object")
    dPrint("selected: ")
    selObjects()
    dPrint("==== return =====")

def lightingOn():

```

```

dPrint("Sun layer 0 true")
bpy.data.objects["Sun"].layers[0] = True
dPrint("Lamp layer 0 true")
bpy.data.objects["Lamp"].layers[0] = True
dPrint("Spot layer 0 true")

print(bpy.data.objects["Spot"].layers[0])
bpy.data.objects["Spot"].layers[0] = True
print(bpy.data.objects["Spot"].layers[0])

#deliberate error to flush display weird
#bpy.ops.wm.safe_mainfile(check_existing=False)

#bpy.ops.wm.save_mainfile(check_existing=False)

def pauseForUser():
    user_select = int(input("enter some user_select: "))
    print("you entered the following user_select: ")
    print(str(user_select))
    print(type(user_select))

#*****
def lightingOff():
#*****
    dPrint("Sun layer 0 false")
    bpy.data.objects["Sun"].layers[0] = False
    dPrint("Lamp layer 0 false")
    bpy.data.objects["Lamp"].layers[0] = False
    dPrint("Spot layer 0 false")
    print(bpy.data.objects["Spot"].layers[0])
    bpy.data.objects["Spot"].layers[0] = False
    print(bpy.data.objects["Spot"].layers[0])
    #bpy.ops.wm.safe_mainfile(check_existing=False)

#*****
def readfile():
#*****
    status = "readFile"
    print("current directory...")
    print(os.getcwd())
    print("changing to C:\\tmp")
    os.chdir("C:\\tmp")
    print("current directory...")
    print(os.getcwd())
    f = open("richfile.txt", 'r')

```

```

clist = f.readlines()
linecount = 0
#stat_strng[1] = "readin scene header"
print(status)
status = stat_strng[1]
print("after ")
print(status)

#read in successive lines from the input file
for c in clist:
    linecount = linecount + 1
    print("line: " + str(linecount) + " = " + c)
    print(c)
#pat = "([INT.][EXT.])(:)([a-c]*)([x-z]*)"
print("status, stat_strng")
print(status)
print(stat_strng[1])
if (status == stat_strng[1]):
    #stat[1] = "readin scene header"
    pat = scplay_exp[1]
    print("pat, c")
    print(pat + " , " + c)
    dPrint( "findit = re.match( " + pat + " , " + c + " )")
    findit = re.match(pat, c)
    if findit:
        status = stat_strng[2]
        #looking for interior/exterior, day/night, type of house
        #scplay_exp = ["dummy", "(INT.|EXT.)([a-
z+)(DAY|NIGHT)", "(runto.|runfrom.)([a-z+)(STRAIGHT|ZIGZAG)"]
        #stat_strng = ["dummy", "readin scene header", "readin
action1"]

        print("groups(): ")
        for g in findit.groups(0):
            print(g)
        print("int or ext is: " + findit.groups(0)[0])
        print("location is : " + findit.groups(0)[1])
        print("day or night : " + findit.groups(0)[2])
        #----- type of house -----
        # select appropriate house from object list
        # and make appropriate house visible in layer 1...
        # deselect all objects first because 2.5 bug (extend=False
fails)

        bpy.ops.object.select_all(action="DESELECT")

        housename = findit.groups(0)[1]

```

```

try:
    houseobj = bpy.data.objects[housename]
except:
    print("ReadFile() - Object: " + housename + "
doesn't exist")
    return

houseobj.layers[0] = True
bpy.ops.object.select_name(name=housename,
extend=False)

#if (findit.groups(0)[1] == "bleakhouse"):
## print("setting layers 0 to true")
### print("readfile, selecting bleakhouse")
#### bpy.data.objects["bleakhouse"].layers[0] = True
##### bpy.ops.object.select_name(name="bleakhouse",
extend=False)

##### pauseForUser()
#####elif (findit.groups(0)[1] == "happyhouse"):
##### bpy.data.objects["happyhouse"].layers[0] =
True

##### print("readfile, selecting happyhouse")
#####
bpy.ops.object.select_name(name="happyhouse", extend=False)
#####else:
##### print("(findit.groups(0)[1] not
recognizable!!!")

# make copies of the appropriate house (will copy selected
object)

if (len(bpy.context.selected_objects) > 0):
    bpy.context.selected_objects[0].layers[0]=True
    copyHouses(housename)
else:
    print("read file, no objects have been selected?")
    print("(nothing in Blender selected_objects_list, no:
")

    print(" object named: " + findit.groups(0)[1])
#----- day or night -----
if (findit.groups(0)[2] == "DAY"):
    print("add lighting to this layer")
    lightingOn()
    #####bpy.data.objects["Sun"].layers[0] = True
    #####bpy.data.objects["Lamp"].layers[0] = True

```

```

        ##bpy.data.objects["Spot"].layers[0] = True
    elif (findit.groups(0)[2] == "NIGHT"):
        print("remove lighting from this layer")
        lightingOff()
        #bpy.data.objects["Sun"].layers[0] = False
        ##bpy.data.objects["Lamp"].layers[0] = False
        ###bpy.data.objects["Spot"].layers[0] = Fals
    else:
        print("(findit.groups(0)[2] not recognizable!!!")
        #-----
elif (status == stat_strng[2]):
    #stat_string[2] = "readin action1 "
    pat = scplay_exp[2]
    findit = re.match(pat, c)
    if findit:
        status =stat_strng[1]
        for g in findit.groups(0):
            print(g)
        print("runto or runfrom is: " + findit.groups(0)[0])
        print("location is : " + findit.groups(0)[1])
        print("straight or zigzag : " + findit.groups(0)[2])
    else:
        print("parser not finding any valid patterns:")
        for i in range(1,2):
            print("-----")
            print(scplay_exp[i])
            print(stat_strng[i])

def idata():
    for o in bpy.data.objects:
        print(o.id_data.data)

def listObjects():
    i = 0
    tx1 = tx2 = tx3 = tx4 = tx5 = ""
    print("this should list objects/indeces")
    for obs in bpy.data.objects:
        tx1 = ""
        tx1 = tx1.join([str(i), ": ", obs.name])
        print(tx1)
        i = i + 1

#xxx go here

```

```

bpy.data.scenes["Scene"].frame_set(1)

def rotBone(name, degree):
    bpy.ops.object.mode_set(mode="POSE")
    bpy.ops.pose.select_all(action="DESELECT")
    bpy.data.armatures["Armature"].bones[name].select = True
    bpy.ops.transform.rotate(value=(degree,), axis = (0,1,0))
    bpy.ops.pose.select_all(action="DESELECT")

def poseAnim():
    ##### need to turn this into action walkcycle rather than repeat rotations for whole path:
    future fix
    bpy.data.scenes["Scene"].frame_set(1)
    rotBone("Bone_R.004", .7)
    rotBone("Bone_R.005", .5)
    rotBone("Bone_L.004", -.9)
    rotBone("Bone_L.005", .5)
    rotBone("Bone_R.001", -.7)
    rotBone("Bone_R.002", -1.8)
    rotBone("Bone_L.001", .7)
    rotBone("Bone_L.002", -.5)
    bpy.ops.pose.copy()
    bpy.ops.pose.select_all(action="SELECT")
    bpy.ops.anim.keyframe_insert_menu(type="Rotation")

    for frameNum in range(1,11):
        bpy.data.scenes["Scene"].frame_set(frameNum*10)
        bpy.ops.pose.paste(flipped=True)
        bpy.ops.pose.select_all(action="SELECT")
        bpy.ops.anim.keyframe_insert(type="Rotation")
        bpy.ops.pose.copy()

def bratAnim():
    name = "Armature"
    ktype = "LocRotScale"

    #reset body to origin, original rotation, just in case, and make it active/selected
    object
    c = bpy.data.objects[name]
    c.rotation_euler = BODY_ROT
    c.location = mathutils.Vector((20,20,0))
    selectOne(name)
    bpy.data.scenes["Scene"].frame_current = 1
    c.keyframe_insert("location")
    #c.keyframe_insert("rotation")

```

```

#bodyPath = [(0,2), (1,3), (1,-3), (0,4), (1,5), (0,-2), (1,-8), (0,-4), (1,4), (0,-3)]
for frameNum in range(1,11):
    axis = (random.sample(range(0,2),1))[0]
    len= (random.sample(range(0,7),1))[0]
    len *= -1
    #axis = bodyPath[frameNum-1][0]
    #len = bodyPath[frameNum-1][1]
    moveBody(name,axis, len)
    bpy.data.scenes["Scene"].frame_current = int(frameNum*10)
    c.keyframe_insert("location")
    #c.keyframe_insert("rotation")

#animate the walk "cycle" we actually keep repeating keyframes instead of cycles
#future fix...
poseAnim()

bpy.data.scenes["Scene"].frame_start = 1
bpy.data.scenes["Scene"].frame_end = 100
#bpy.context.scene.render.file_format = 'FFMPEG'
#bpy.ops.render.render(animation=True)
bpy.ops.screen.animation_play()

def selObjects():
    if (debugLevel["selObjects"] != 1):
        return
    sl = len(bpy.context.selected_objects)
    if (sl <= 0):
        print("selObjects() {<= 0} and/or NOTHING SELECTED!")
        return()
    else:
        print("There are " + str(sl) + " selected objectss: ")
        for o in bpy.context.selected_objects:
            print(o.name)

#*****
def buildHouse(name, combo):
#*****

    dPrint("Entering buildHouse(name=<" + name + ">, combo=<" + str(combo)+">")

    selectOne("Camera")
    bpy.ops.object.select_name(name="Lamp", extend=True)
    bpy.ops.object.select_inverse()
    bpy.ops.object.delete()
    bpy.ops.mesh.primitive_cube_add()

```

```

dPrint( "calling vef_mode(face)" )
vef_mode("face")

for level in hlevels:
#
# build each level/bodyPart in loop, a complete house/body is built out of levels
(face, uppr, roof, etc...)
# -----
# each level/bodyPart is essentially a cube and will be textured from a user
defined 2D drawing
# using the "cube_project" uv mapping method. After texturing a "face" cube,
"uppr" cube,
# "roof" cube, etc... the building blocks can be put together into a house and then
copied

print("***** buildHouse: (" + name + ") *****")
print("starting level loop: level = " + level)

matexs(" 100 ")
houselevel = name + "house" + level

# each level/building block of house starts out as cube or plane
# add a cube (or 'plane' for the street level) primitive mesh
# first set to OBJECT mode, but mode_set does not work if no MESH's
objects
try:
    bpy.ops.object.mode_set(mode='OBJECT')
    bpy.ops.object.select_all(action="DESELECT")
except:
    print("error setting OBJECT mode (no MESH objects?)")

if (level != "stret"):
    bpy.ops.mesh.primitive_cube_add()
else:
    bpy.ops.mesh.primitive_plane_add()

#give it the name of the level
bpy.context.selected_objects[0].name = level
bpy.ops.object.mode_set(mode='OBJECT')
bpy.ops.object.select_all(action="DESELECT")
c = bpy.data.objects[level]
c.select = True

#make sure at least one slot in ...objects[level]...material_slots[]

```



```

bpy.ops.object.material_slot_add()
matexs(" 200 ")

#make sure at least one material["Material"] in bpy.data.materials[]
bpy.ops.material.new()
matexs(" 300 ")

#new material default name is "Material" change to level
bpy.data.materials["Material"].name = level
matexs(" 400 ")

#make the slot point to the material (slot(0) of current obj)0
c.material_slots[0].material = bpy.data.materials[level]
matexs(" 500 ")

#make sure at least one texture["Texture"] in bpy.data.textures
bpy.ops.texture.new()
#new texture default name is "Texture" change to level
bpy.data.textures["Texture"].name = level
matexs(" 600 ")

# any material can get up to (18?) texture slots
# we will enable the first texture slot for above created material
# and have it point to the newly created (above) textures[level]
c.material_slots[0].material.texture_slots.add()
c.material_slots[0].material.texture_slots[0].texture =
bpy.data.textures[level]
c.material_slots[0].material.texture_slots[0].texture.type = "IMAGE"
# need a repeat factor for these level(s): street, ...
if (level == "stret"):
    bpy.data.textures[level].repeat_x = cobblestone_x_rep
    bpy.data.textures[level].repeat_y = cobblestone_y_rep
matexs(" 700 ")

housenamelevelpath = paths["drawings"] + housenamelevel + ".png"
print("opening file: housenamelevelpath = ", housenamelevelpath)
try:
    bpy.ops.image.open(filepath = housenamelevelpath)
except:
    print("ERROR! (returning out) with bpy.ops.image.open(filepath =
" + housenamelevelpath + ")")
    return

c.material_slots[0].material.texture_slots[0].texture.image =
bpy.data.images[housenamelevel + ".png"]

```

```

        if (level == "head" or level == "arm1" or level == "leg1"
            or level == "arm2" or level == "leg2" or level == "body"):
            selectSideFaces()
#for these parts no texture on top or bottom

        bpy.ops.object.mode_set(mode='EDIT')

        if (level == "face" or level == "uppr" or level == "roof"):
            dPrint("cube_project: " + level)
            bpy.ops.uv.cube_project()
        elif (level == "stret" or level == "head" or level == "arm1" or level ==
"leg1"
            or level == "arm2" or level == "leg2" or level == "body"):
            dPrint("smart_project: " + level)
            bpy.ops.uv.smart_project()

        c.material_slots[0].material.texture_slots[0].texture_coords = 'UV'
        matexs(" 800 ")

        # get image attributes and adjust cube accordingly
        # ***** FIX !!!!!
        xdimension = sizes[level][0]
        ydimension = sizes[level][1]
        zdimension = sizes[level][2]
        bpy.data.objects[level].dimensions = mathutils.Vector((xdimension,
ydimension, zdimension))

        #put body in diff layer to avoid clutter
        if level in bodyPartNames:
            dPrint("setting layers(4) for " + level + " to true")
            dPrint("layers[0], layers[4] = " +
str(bpy.data.objects[level].layers[0])
                + " " + str(bpy.data.objects[level].layers[4]))
            bpy.data.objects[level].layers[4] = True
            #place body part in appropriate position
            bpy.data.objects[level].location = positions[level]
            for i in range(3):
                dPrint("\n setting location for: " + level + " to: " +
str(positions[level][i]))

            #to create elbow and knee joint extra cubes...
            #upper arm and upper leg cubes are dup'ed several times, with each
dup shrink z axis
            #then offset downward, creates a stack of very thin slightly offset

```

```

cubes for knee/elb joint
    if (level == "arm1" or level == "leg1"):
        for i in range(1,4):
            selectOne(level)
            bpy.ops.object.mode_set(mode='OBJECT')

            bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"linked":False,
"mode":'TRANSLATION'},
            TRANSFORM_OT_translate={"value":(0, 0, 0),
            "constraint_axis":(False, False, True),
            "constraint_orientation":'GLOBAL',
            "mirror":False, "proportional":'DISABLED',
            "proportional_edit_falloff":'SMOOTH',
            "proportional_size":1, "snap":False,
            "snap_target":'CLOSEST', "snap_point":(0,
0, 0), "snap_align":False, "snap_normal":(0, 0, 0),
            "texture_space":False,
            "release_confirm":False})
            bpy.ops.transform.resize(value=(-0.062647, -
0.062647, -0.062647),
            constraint_axis=(False, False, True),
            constraint_orientation='GLOBAL', mirror=False,
            proportional='DISABLED',
            proportional_edit_falloff='SMOOTH', proportional_size=1, snap=False,
            snap_target='CLOSEST', snap_point=(0, 0, 0),
            snap_align=False, snap_normal=(0, 0, 0),
            texture_space=False, release_confirm=False)
            jointName = level + "_" + str(i)
            bpy.context.selected_objects[0].name = jointName
            zmove1 = -1.03 * (
bpy.data.objects[level].dimensions[2]/2 + bpy.data.objects[jointName].dimensions[2]/2 )
            zmove = zmove1 + (-1.10 * ((i-
1)*(bpy.data.objects[jointName].dimensions[2])) )
            bpy.ops.transform.translate(value=(0, 0, zmove),
            constraint_axis=(False, False, True),
            constraint_orientation='GLOBAL', mirror=False,
            proportional='DISABLED',
            proportional_edit_falloff='SMOOTH',
            proportional_size=1, snap=False,
            snap_target='CLOSEST', snap_point=(0, 0, 0),
            snap_align=False, snap_normal=(0, 0, 0),
            texture_space=False, release_confirm=False)
            bpy.ops.object.select_all(action = "DESELECT")
            selectOne(level)
            bpy.data.objects[jointName].select = True

```

```

                                #bpy.ops.object.join()

                                #subdivide and delete half of head/body so we can add y axis
symmetry
                                #will be helpful when armature used to animate it
                                if (level == "body" or level == "head"):
                                    selectOne(level)
                                    bpy.ops.object.mode_set(mode='EDIT')
                                    bpy.ops.mesh.select_all(action="SELECT")
                                    bpy.ops.mesh.subdivide(number_cuts=1, smoothness=0,
                                                                fractal=0,
corner_cut_pattern='INNER_VERTEX')
                                    bpy.ops.mesh.select_all(action="DESELECT")

                                    bpy.ops.object.mode_set(mode='OBJECT')
                                    for face in bpy.context.active_object.data.faces:
                                        print("face index", face.index)
                                        if face.center[1] < .001:
                                            face.select = True

                                    bpy.ops.object.mode_set(mode='EDIT')
                                    bpy.ops.mesh.delete(type="FACE")

#=====
#-----#
#----- end of looping through house levels/cubes
#----- we now have cubes for (face, uppr, roof, etc...)
#=====
#-----#

# move roof to top of face, make street plane larger than house base, etc...
rh = bpy.data.objects["roof"].dimensions.z
fh = bpy.data.objects["face"].dimensions.z
bpy.data.objects["roof"].location[2] = fh/2 + rh/2
bpy.data.objects["stret"].location[2] = bpy.data.objects["face"].location[2] - fh/2 -
.01
bpy.data.objects["stret"].dimensions = mathutils.Vector((3,3,0)) # 3 no work for
3d texture view
#bpy.data.objects["stret"].dimensions.y = 9 # 3 no work for 3d texture view

upstring = "0000 Alpha print bug"
print(upstring)

# create the levels necessary for a 2 story version (requires "uppr" level)
selectOne("roof")

```

```

upstring = "000 Bravo print bug"
print(upstring)
zroof = fh
upstring = "000 Charlie print bug"
print(upstring) #cannot print anything after dup of "roof" WHY??????
bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"linked":False,
"mode":1}, TRANSFORM_OT_translate={"value":(3, 3,
zroof),"release_confirm":False})
upstring = "000 Delta print bug"
#print(upstring)
bpy.context.selected_objects[0].name = "roof_2flr"

upstring = "1111 print bug"
#print(upstring)

selectOne("uppr")
zupper = fh
bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"linked":False,
"mode":1}, TRANSFORM_OT_translate={"value":(3, 3,
zupper),"release_confirm":False})
bpy.context.selected_objects[0].name = "uppr_2flr"
#cannot do here, end up with no active object then selectOne doesn't work
#selectOne("uppr")
#bpy.ops.object.delete()

upstring = "2222 print bug"
#print(upstring)

selectOne("face")
bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"linked":False,
"mode":1}, TRANSFORM_OT_translate={"value":(3, 3, 0),"release_confirm":False})
bpy.context.selected_objects[0].name = "face_2flr"

selectOne("stret")
bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"linked":False,
"mode":1}, TRANSFORM_OT_translate={"value":(3, 3, 0),"release_confirm":False})
bpy.context.selected_objects[0].name = "stret_2flr"

selectOne("uppr")
bpy.ops.object.delete()

if (combo):
    combineLevels(name)

if (COMBINE_BODY):

```

```

        combineBody()
        bpy.ops.object.modifier_add(type="MIRROR")
        bpy.data.objects["body"].modifiers["Mirror"].use_x = False
        bpy.data.objects["body"].modifiers["Mirror"].use_y = True

dPrint("BUILD_ARMATURE = " + str(BUILD_ARMATURE))
if (BUILD_ARMATURE):
    dPrint("BUILD_ARMATURE = " + str(BUILD_ARMATURE))
    buildArmature()
    #parent body to the armature ("skin" it)
    selectOne("body")
    bpy.ops.object.select_name(name="Armature", extend=True)
    bpy.ops.object.parent_set(type="ARMATURE_AUTO")

if (CREATE_PATH):
    createPath()
    pathParent()

#*****
def combineLevels(name):
#*****
    rh = bpy.data.objects["roof"].dimensions.z
    fh = bpy.data.objects["face"].dimensions.z

    dPrint("combineLevels selecting roof")
    print("combineLevels selecting roof")
    selectOne("roof")
    bpy.ops.object.select_name(name="face", extend=True)
    bpy.ops.object.select_name(name="stret", extend=True)
    bpy.ops.object.join()
    comboObj = bpy.context.selected_objects[0]
    two_print("setting 1 floor " + comboObj.name + " name to " + name)
    comboObj.name = name

    dPrint("combineLevels selecting roof_2flr")
    print("combineLevels selecting roof_2flr")
    selectOne("roof_2flr")
    bpy.data.objects["uppr_2flr"].select = True
    bpy.data.objects["face_2flr"].select = True
    bpy.data.objects["stret_2flr"].select = True
    bpy.ops.object.join()
    comboObjTall = bpy.context.selected_objects[0]

```

```

print("setting 2 floor " + comboObjTall.name + " name to " + name)
dPrint("xxxsetting 1 floor " + comboObj.name + " name to " + name)
comboObjTall.name = name + "_2flr"

#scaleHouse = 1.6
#comboObj.scale = mathutils.Vector(( scaleHouse, scaleHouse, scaleHouse ))
#comboObjTall.scale = mathutils.Vector(( scaleHouse, scaleHouse, scaleHouse ))

##### create the base plane, flat terrain plane under everything else
#####
bpy.ops.object.mode_set(mode='OBJECT')
bpy.ops.object.select_all(action="DESELECT")
bpy.ops.mesh.primitive_plane_add()
bp = bpy.context.selected_objects[0]
bp.name = "basePlane"
bp.dimensions = mathutils.Vector((4700, 600, 0))
bp.scale = mathutils.Vector((-2400, 30, 1))
bp.location = mathutils.Vector((13, 7, -1.5))

#####
def combineBody():
#####
    merge_joint()
    two_print("combineLevels selecting head, armfour, legfour")
    selectOne("head")
    for bodyPart in ["armfour", "legfour"]:
        bpy.data.objects[bodyPart].select = True
    bpy.ops.object.select_name(name = "body", extend=True)
    bpy.ops.object.join()
    comboObjBody = bpy.context.selected_objects[0]
    comboObjBody.name = "body"
    selectOne("body")
    bpy.ops.transform.rotate(value=(-1.5708,), axis=(0, 0, 1),
constraint_axis=(False, False, True),
        constraint_orientation='GLOBAL', mirror=False,
proportional='DISABLED', proportional_edit_falloff='SMOOTH',
        proportional_size=1, snap=False, snap_target='CLOSEST',
snap_point=(0, 0, 0), snap_align=False,
        snap_normal=(0, 0, 0), release_confirm=False)

def boneTest():
    fromName = "Bone"
    toName = "toName"
    ignoreThese = []

```

```

renames = 0
for chld in list(bpy.data.armatures["Armature"].bones[fromName].children):
    if chld.name not in ignoreThese:
        chld.name = toName
        renames = renames + 1
        if renames > 1:
            print("ERROR!")

def extrBone(fromName, toName, ignoreThese, tranVec=(1,1,1), constraint=(False,
False, False)):
    #extrBone(fromName, toName, ignoreThese, tranVec=(1,1,1), constraint=(False,
False, False)):
    bpy.ops.armature.select_all(action="DESELECT")
    bpy.context.object.data.edit_bones[fromName].select_tail = True
    bpy.ops.armature.extrude_forked(ARMATURE_OT_extrude={"forked":False},
    TRANSFORM_OT_translate={"value":tranVec,"constraint_axis":(False, False,
False),
    "constraint_orientation":'GLOBAL', "mirror":False,
"proportional":'DISABLED',
    "proportional_edit_falloff":'SMOOTH', "proportional_size":1,
    "snap":False, "snap_target":'CLOSEST', "snap_point":(0, 0, 0),
"snap_align":False,
    "snap_normal":(0, 0, 0), "texture_space":False,
    "release_confirm":False})
    bpy.ops.object.mode_set(mode='OBJECT')
    bpy.ops.object.mode_set(mode='EDIT')
    renames = 0
    for chld in list(bpy.data.armatures["Armature"].bones[fromName].children):
        if chld.name not in ignoreThese:
            two_print("renaming bone: " + chld.name)
            two_print("  new name:  " + toName)
            chld.name = toName
            renames = renames + 1
            if renames > 1:
                two_print("extrBone: ignoreThese list needs to list ALL
children!")
                two_print("some children improperly renamed")

    index = len(bpy.data.armatures["Armature"].bones)-1
    #bpy.context.object.data.edit_bones[index].name = toName

def clearEditBones():
    bnz = bpy.context.object.data.edit_bones
    for bone in bnz:
        bone.select = False

```



```

bone.select_head = False
bone.select_tail = False

def selAllParts(bone_name):
    bnz = bpy.context.object.data.edit_bones
    bnz[bone_name].select = True
    bnz[bone_name].select_head = True
    bnz[bone_name].select_tail = True

def moveIt(name):
    bnz = bpy.context.object.data.edit_bones
    clearEditBones()
    selAllParts(name)
    bpy.ops.armature.parent_clear(type="DISCONNECT")
    selAllParts(name + ".001")
    selAllParts(name + ".002")
    bpy.ops.transform.translate(value=(0.0228726, 1.59803e-007, -1.18938),
    constraint_axis=(False, False, False), constraint_orientation='GLOBAL', mirror=False,
    proportional='DISABLED', proportional_edit_falloff='SMOOTH', proportional_size=1,
    snap=False, snap_target='CLOSEST', snap_point=(0, 0, 0), snap_align=False,
    snap_normal=(0, 0, 0), texture_space=False, release_confirm=False)

def armat2():
    #There is a bug, mirrored bones not created correctly if try to use _mirror_x
    #programatically through API
    #need user interactive intervention between each bone creation to cause mirror
    #bone to appear etc...
    #so unfortunately we need to manually build right and left sides, then make sure
    #name convention followd
    boneVec1 = sizes["leg1"][2] * (-1.1) # bone must account for joint space
    boneVec2 = sizes["leg2"][2] * (-1.1) # bone must account for joint space
    bpy.data.armatures["Armature"].use_mirror_x = False
    #Right and Left leg bones
    extrBone("Bone", "Bone_R", [], (.35, 0, 0))
    extrBone("Bone", "Bone_L", ["Bone_R"], (-.35, 0, 0))
    extrBone("Bone_R", "Bone_R.001", [], (0, 0, boneVec1))
    extrBone("Bone_L", "Bone_L.001", [], (0, 0, boneVec1))
    extrBone("Bone_R.001", "Bone_R.002", [], (0, 0, boneVec2))
    extrBone("Bone_L.001", "Bone_L.002", [], (0, 0, boneVec2))

def legPlace():
    #move down to leg position
    moveIt("Bone_R")
    moveIt("Bone_L")

```

```

def armBones():
    boneVec1 = sizes["arm1"][2] * (-1.1) # bone must account for joint space
    boneVec2 = sizes["arm2"][2] * (-1.1) # bone must account for joint space
    #as previously mentioned, mirror works with interactive UI ops, but not with API
calls :-( what a headache!
    bpy.data.armatures["Armature"].use_mirror_x = False
    #Right and left arm bones (U = Upper = arm)
    extrBone("Bone", "Bone_R.003", ["Bone_R", "Bone_L"], (.85, 0, 0))
    extrBone("Bone", "Bone_L.003", ["Bone_R.003", "Bone_R", "Bone_L"], (-.85,
0, 0))
    extrBone("Bone_R.003", "Bone_R.004", [], (0, 0, boneVec1))
    extrBone("Bone_L.003", "Bone_L.004", [], (0, 0, boneVec1))
    extrBone("Bone_R.004", "Bone_R.005", [], (0, 0, boneVec2))
    extrBone("Bone_L.004", "Bone_L.005", [], (0, 0, boneVec2))

def buildArmature():
    bpy.ops.object.armature_add(view_align=False, enter_editmode=False,
location=(0, 0, -0.5),
    rotation=(0, 0, 0), layers=(True, False, False, False, True, False, False, False,
False, False,
    False, False, False, False, False, False, False, False, False, False))
    bpy.ops.transform.translate(value=(0, 0, -0.458134), constraint_axis=(False,
False, True), constraint_orientation='GLOBAL',
    mirror=False, proportional='DISABLED', proportional_edit_falloff='SMOOTH',
proportional_size=1, snap=False,
    snap_target='CLOSEST', snap_point=(0, 0, 0), snap_align=False,
snap_normal=(0, 0, 0), texture_space=False,
    release_confirm=False)
    bpy.ops.object.mode_set(mode='EDIT')
    armat2()
    legPlace()
    armBones()

def movToLeg():
    selectOne("Bone_R.001")
    bpy.ops.armature.parent_clear(type='CLEAR')
    bpy.ops.transform.translate(value=(-0.639964, 1.7197e-007, -1.27993),
constraint_axis=(False, False, False),
    constraint_orientation='GLOBAL', mirror=False, proportional='DISABLED',
proportional_edit_falloff='SMOOTH',
    proportional_size=1, snap=False, snap_target='CLOSEST', snap_point=(0, 0,
0), snap_align=False,
    snap_normal=(0, 0, 0), texture_space=False, release_confirm=False)

#*****

```

```

def copyHouses(name, perc, row_cols, street_modulo):
#*****

    dPrint("in copyHouses(name=<"+name+">, perc=<"+str(perc)+">,
row_cols=<"+str(row_cols)+
    ">, street_modulo=<"+str(street_modulo)+">")

    if (name=="levels"):
        # we are copying individual levels of house separately
        print("will only copy 1 floor levels that were not combined")
        for level in hlevels:
            copyObject(bpy.data.objects[level], row_cols, street_modulo)
    else:
        copyObject(bpy.data.objects[name], perc, row_cols, street_modulo)

#*****
# functions to join joint-cubes and merge vertices
#*****

#*****
def fixModeSet(contextMode):
#*****
    # this is a fix for the problem where the strings returned by context.mode
    # do not match the (string) input arguments for ops.object.mode_set()
    if (contextMode == "EDIT_MESH"):
        contextMode = "EDIT"
    elif (contextMode == "PAINT_VERTEX"):
        contextMode = "VERTEX_PAINT"
    bpy.ops.object.mode_set(mode=contextMode)

#*****
def merge_joint():
#*****
    jjoin("arm1", "arm1_1", "armone")
    merge_couple("armone",0,8)
    merge_couple("armone",0,8)
    merge_couple("armone",0,8)
    merge_couple("armone",0,8)

    jjoin("armone", "arm1_2", "armtwo")
    merge_couple("armtwo",1,9)
    merge_couple("armtwo",0,7)
    merge_couple("armtwo",1,9)
    merge_couple("armtwo",0,7)

```

```
jjoin("armtwo", "arm1_3", "armthree")
merge_couple("armthree",1,9)
merge_couple("armthree",1,9)
merge_couple("armthree",1,9)
merge_couple("armthree",0,5)
```

```
jjoin("armthree", "arm2", "armfour")
merge_couple("armfour",5,9)
merge_couple("armfour",5,9)
merge_couple("armfour",5,9)
merge_couple("armfour",4,5)
```

```
jjoin("leg1", "leg1_1", "legone")
merge_couple("legone",0,8)
merge_couple("legone",0,8)
merge_couple("legone",0,8)
merge_couple("legone",0,8)
```

```
jjoin("legone", "leg1_2", "legtwo")
merge_couple("legtwo",1,9)
merge_couple("legtwo",0,7)
merge_couple("legtwo",1,9)
merge_couple("legtwo",0,7)
```

```
jjoin("legtwo", "leg1_3", "legthree")
merge_couple("legthree",1,9)
merge_couple("legthree",1,9)
merge_couple("legthree",1,9)
merge_couple("legthree",0,5)
```

```
jjoin("legthree", "leg2", "legfour")
merge_couple("legfour",5,9)
merge_couple("legfour",5,9)
merge_couple("legfour",5,9)
merge_couple("legfour",4,5)
```

```
def jjoin(joint1, joint2, new_name):
    #join 2 joints and give new ob given name
    saveMode = bpy.context.mode
    bpy.ops.object.mode_set(mode='OBJECT')
    selectOne(joint1)
    bpy.ops.object.select_name(name=joint2, extend=True)
    two_print("joining 2 joints: " + joint1 + ", " + joint2)
```

```

bpy.ops.object.join()
bpy.context.active_object.name = new_name
fixModeSet(saveMode)

def clear1():
    saveMode = bpy.context.mode
    print("====selected objects, before: selectOne(arm1)")
    selObjects()
    selectOne("arm1")
    print("====selected objects, after: selectOne(arm1)")
    selObjects()
    print("====selected objects, after calling select_all(DESELECT)")
    bpy.ops.object.mode_set(mode='OBJECT')
    bpy.ops.object.select_all(action= 'DESELECT')
    selObjects()
    fixModeSet(saveMode)

#*****
def merge_couple(obname, vidx1, vidx2):
#*****
    clearv1(obname)
    pickv1(obname, vidx1, vidx2)
    merge1()

#*****
#def vertEdgeFaceMode(mode):
def vef_mode(mode="nochange"):
#*****
    saveMode = bpy.context.mode
    bpy.ops.object.mode_set(mode='EDIT')

two_print("in vef_mode, current mesh select mode settings...")
two_print(str(list(bpy.context.tool_settings.mesh_select_mode)))
two_print("in vef_mode(...) setting mode to " + mode)

if (mode == "nochange"):
    pass
elif (mode == "vertex"):
    bpy.context.tool_settings.mesh_select_mode[0] = True
    bpy.context.tool_settings.mesh_select_mode[1] = False
    bpy.context.tool_settings.mesh_select_mode[2] = False
elif (mode == "edge"):
    bpy.context.tool_settings.mesh_select_mode[1] = True

```

```

    bpy.context.tool_settings.mesh_select_mode[0] = False
    bpy.context.tool_settings.mesh_select_mode[2] = False
elif (mode == "face"):
    bpy.context.tool_settings.mesh_select_mode[2] = True
    bpy.context.tool_settings.mesh_select_mode[0] = False
    bpy.context.tool_settings.mesh_select_mode[1] = False
else:
    two_print("edgeVertFaceMode, invalid mode = " + str(mode))

two_print("exiting vef_mode, current mesh select mode settings...")
two_print(str(list(bpy.context.tool_settings.mesh_select_mode)))

fixModeSet(saveMode)

```

```

#=====
def clearv1(obname):
#=====
    saveMode = bpy.context.mode
    selectOne(obname)
    #setting vert.select does not work unless
    #edit is in vertex mode
    bpy.ops.object.mode_set(mode='EDIT')
    bpy.context.tool_settings.mesh_select_mode[0] = True
    bpy.context.tool_settings.mesh_select_mode[1] = False
    bpy.context.tool_settings.mesh_select_mode[2] = False
    bpy.ops.object.mode_set(mode='OBJECT')
    for vert in bpy.data.objects[obname].data.vertices:
        vert.select = False
    fixModeSet(saveMode)

def listv1(obname=None):
    if obname == None:
        obname = bpy.context.active_object.name
    saveMode = bpy.context.mode
    bpy.ops.object.mode_set(mode='OBJECT')
    two_print("*** all vertices ===")
    for vert in bpy.data.objects[obname].data.vertices:
        two_print(str(vert.index) + "\t" + str(vert.select))
    two_print("*** selected vertices ===")
    for vert in bpy.data.objects[obname].data.vertices:
        if vert.select == True:
            two_print(str(vert.index) + "\t" + str(vert.select))
    fixModeSet(saveMode)

```

```

def pickv1(obname, vidx1, vidx2):
    saveMode = bpy.context.mode
    clearv1(obname)
    bpy.ops.object.mode_set(mode='OBJECT')
    bpy.data.objects[obname].data.vertices[vidx1].select = True
    bpy.data.objects[obname].data.vertices[vidx2].select = True
    fixModeSet(saveMode)

def merge1():
    saveMode = bpy.context.mode
    bpy.ops.object.mode_set(mode='EDIT')
    bpy.context.tool_settings.mesh_select_mode[0] = True
    bpy.context.tool_settings.mesh_select_mode[1] = False
    bpy.context.tool_settings.mesh_select_mode[2] = False
    bpy.ops.mesh.merge(type='CENTER', uvs=False)
    fixModeSet(saveMode)

#*****
def listvfs():
#*****
    current_obj = bpy.context.active_object
    selectOne(current_obj.name)
    bpy.ops.object.mode_set(mode='OBJECT')

    for vert in current_obj.data.vertices:
        print( "vertex index(" + str(vert.index) + ")" )
        if (vert.select):
            print( "  selected(" + str(vert.select) + ")" )
        else:
            print( "  ***** selected(" + str(vert.select) + ")" )

    for face in current_obj.data.faces:
        print( "face index(" + str(face.index) + ")" )
        if (face.select):
            print( "  selected(" + str(face.select) + ")" )
        else:
            print( "  ***** selected(" + str(face.select) + ")" )

#*****
def selectSideFaces():
#*****
    current_obj = bpy.context.active_object
    selectOne(current_obj.name)

    bpy.ops.object.mode_set(mode='EDIT')

```

```

bpy.ops.mesh.select_all(action= 'DESELECT')
bpy.ops.object.mode_set(mode='OBJECT')

for face in current_obj.data.faces:
    two_print( "before selectSideFaces...")
    two_print( "face index(" + str(face.index)  + ")" )
    two_print( "face select(" + str(face.select)  + ")" )
    if face.index > 1 and current_obj.name != "noclear":
        face.select = True
    two_print( "aftsaer selectSideFaces...")
    two_print( "face index(" + str(face.index)  + ")" )
    two_print( "face select(" + str(face.select)  + ")" )

#*****
def shapePath():
#*****
    print("in shapePath")
    saveMode = bpy.context.mode
    bpy.ops.object.mode_set(mode='EDIT')
    pathPoints = [(5,0,0),(0,2,0),(3,0,0),(0,4,0),(0,-13,0),(6,0,0),(0,11,0),(0,12,0),(-
13,0,0),(10,0,0),(0,-13,0),(6,0,0)]
    for point in pathPoints:
        x = float(point[0] + .01)
        y = float(point[1] + .02)
        z = float(point[2] + .03)
        print("calling extrudePath: " + str(x) + " " + str(y) + " " + str(z))
        bpy.ops.curve.extrude(mode="TRANSLATION")
        bpy.ops.transform.transform(mode="TRANSLATION", value = (x,y,z,0))
    fixModeSet(saveMode)

#*****
def moveBody(name, axis, len):
#*****
    #axis: x = 0, y = 1
    xdelta = 0
    ydelta = 0
    xdelta = (1-axis)*len
    ydelta = axis*len
    if (xdelta > .01):
        pass
        #bpy.data.objects[name].rotation_euler = mathutils.Euler((0.0, -0.0, 0.0),
"XYZ")
    elif (xdelta < -.01):
        pass

```



```

        #bpy.data.objects[name].rotation_euler = mathutils.Euler((0.0, -0.0, 3.14),
"XYZ")
        elif (ydelta > .01):
            pass
            #bpy.data.objects[name].rotation_euler = mathutils.Euler((0.0, -0.0, 1.57),
"XYZ")
        elif (ydelta <- .01):
            pass
            #bpy.data.objects[name].rotation_euler = mathutils.Euler((0.0, -0.0, -1.57),
"XYZ")

        bpy.ops.transform.translate(value=(xdelta, ydelta, 0))

```

```

#*****
def createPath():
#*****
        #creates a NURBS path, but difficult to manage in context of street grid
because NURBS
        #path wants to cut corners, so for now better to use "moveBody(axis, len)" in
straight lines
        saveMode = bpy.context.mode
        bpy.ops.object.mode_set(mode='OBJECT')
        bpy.ops.curve.primitive_nurbs_path_add(view_align=False,
enter_editmode=False,
        location=(0,0,0), rotation=(0, 0, 0),
        layers=(True, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False))
        bpy.ops.transform.translate(value=(2.0, 0, 0),
        constraint_axis=(False, False, False), constraint_orientation='GLOBAL',
        mirror=False, proportional='DISABLED', proportional_edit_falloff='SMOOTH',
        proportional_size=1, snap=False, snap_target='CLOSEST', snap_point=(0, 0, 0),
        snap_align=False, snap_normal=(0, 0, 0), texture_space=False,
        release_confirm=False)
        shapePath()
        fixModeSet(saveMode)

#*****
def pathParent():
#*****
        saveMode = bpy.context.mode
        bpy.ops.object.mode_set(mode='OBJECT')
        selectOne("body")

```

```

bpy.ops.object.select_name(name="NurbsPath", extend=True)
bpy.ops.object.parent_set(type="FOLLOW")
fixModeSet(saveMode)

```

```

#*****
def copyObject(cobj, perc, rc, sm):
#*****
    # build a rc x rc grid of houses (mixture of 2 floor and 1 floor)
    # make a copy of either cobj (1 story house) or cobj2 (2 story house)
    # depending on random factor, and insert it into a city grid of ro/cols
    # (rc x rc) with room for horizontal/vertical streets at the modulo (sm) tics
    #
    print("in copyObject(" + cobj.name + ")")
    tot = int( math.pow(rc, 2) )
    print(type(perc))
    print(type(tot))
    print(type(perc*tot))
    frac = (perc*tot)/100.0
    ifrac = int(frac)
    tallList = random.sample( range(tot), ifrac )
    copy_list = []
    start_loc = cobj.location
    SPACE = cobj.dimensions[0]
    for indx2 in range(rc):
        if not(indx2 % sm):
            continue
        for indx in range(rc):
            if not(indx % sm):
                continue
            if ((indx2*rc)+indx) in tallList:
                selectOne(cobj.name + "_2flr")
            else:
                selectOne(cobj.name)
            dPrint("before duplicate")
            selObjects()
            listObjects()
            dPrint("duplicating for position: " + str(indx2) + " " + str(indx))

    bpy.ops.object.duplicate_move(OBJECT_OT_duplicate={"linked":False,
"mode":'TRANSLATION'}, TRANSFORM_OT_translate={"value":(0, 0, 0),
"constraint_axis":(False, False, False), "constraint_orientation":'GLOBAL',
"mirror":False, "proportional":'DISABLED', "proportional_edit_falloff":'SMOOTH',
"proportional_size":1, "snap":False, "snap_target":'CLOSEST', "snap_point":(0, 0, 0),
"snap_align":False, "snap_normal":(0, 0, 0), "texture_space":False,

```

```

"release_confirm":False})
        dPrint("after duplicate")
        selObjects()
        listObjects()
        so = bpy.context.selected_objects[0]
        so_zloc = so.location[2]
        so.location = start_loc + mathutils.Vector((indx2*SPACE,
indx*SPACE, so_zloc))
        #bpy.ops.transform.translate(value=(0, 4.72359, 0 ),
constraint_axis=(False, True, False), constraint_orientation='GLOBAL', mirror=False,
proportional='DISABLED', proportional_edit_falloff='SMOOTH', proportional_size=1,
snap=False, snap_target='CLOSEST', snap_point=(0, 0, 0), snap_align=False,
snap_normal=(0, 0, 0), release_confirm=False)
        so = bpy.context.selected_objects[0]
        copy_list.append(so)
        l = len(copy_list)
        copy_list[l-1].name = cobj.name + str(l) + "_" + str(indx2) + "_" +
str(indx)
        dPrint(copy_list[l-1].name)

# postion camera (for test results only)
if SET_CAMERA_POSITION_MANUALLY:
    bpy.data.objects["Camera"].rotation_euler[0] = 1.34
    bpy.data.objects["Camera"].rotation_euler[1] = -.07
    bpy.data.objects["Camera"].rotation_euler[2] = -0.8

    bpy.data.objects["Camera"].location[0] = -3.5
    bpy.data.objects["Camera"].location[1] = -1.6
    bpy.data.objects["Camera"].location[2] = 4.3

def modeQuery():
    two_print("in module 'brat.py (function: modeQuery)', __name__ =: " +
__name__)
    two_print(" checking for 'C:\\Blender\\runit*' to determine run mode")
    two_print("  runit.main ----- run main()")
    two_print("  runit.merge_joint -- run merge_joint() ")
    two_print("  runit.auto----- run with hardwired input")
    two_print("  runit.off ----- just import but run nothing (skip main)")
    two_print("  any other or none -- same as runit.off")

    if os.path.isdir("C:\\Blender\\runit.main"):
        return("main")
    elif os.path.isdir("C:\\Blender\\runit.merge_joint"):
        return("merge_joint")
    elif os.path.isdir("C:\\Blender\\runit.auto"):

```

```

        return("auto")
elif os.path.isdir("C:\\Blender\\runit.off"):
    return("off")
else:
    return("off")

def runit():
    user_select = 1
    while (user_select > 0):
        print("0 - exit")
        print("1 - select an object... selectOne(name)")
        print("2 - bratAnim()")
        print("3 - read playwright file()")
        print("4 - turn lighting on")
        print("5 - turn lighting off")
        print("6 - render animation")
        print("7 - copy houses")
        print("8 - list materials/textures")
        print("9 - build house")
        print("10 - selectSideFaces()")

        user_select = int(input("enter choice: "))
        print("you entered the following choice: ")
        print(str(user_select))
        print(type(user_select))
        #print("setting to 2 manually")
        #stuff = 2
        if (user_select == 1):
            print("you entered 1 ... selectOne")
            name = input("enter object name to select")
            selectOne(name)
        if (user_select == 2):
            print("you entered 2 ... bratAnim")
            bratAnim()
        if (user_select == 3):
            print("you entered 3 ... readFile")
            readFile()
        if (user_select == 4):
            print("you entered 4 ... lighting on")
            lightingOn()
        if (user_select == 5):
            print("you entered 5 ... lightin offg")
            lightingOff()
        if (user_select == 6):
            print("you entered 6 ... render animation")

```

```

        renderit()
    if (user_select == 7):
        print("you entered 7 ... copy_houses()")
        name = input("enter combo house name, or 'levels' for level copy")
        perc = int(input("enter percentage of houses with 2 floors"))
        row_cols = int(input("enter number of row/cols for houses: "))
        street_modulo = int(input("enter street modulo: "))
        copyHouses(name, perc, row_cols, street_modulo)
    if (user_select == 8):
        print("you entered 8 ...matexs()")
        old_flag = debugLevel["matexs"]
        debugLevel["matexs"] = 1
        matexs("main menu: matexs()")
        debugLevel["matexs"] = old_flag
    if (user_select == 9):
        print("you entered 9 ... buildHouse()")
        name = input("(No name, enter name of house to build: ")
        if (newWindows == False):
            name = name[0:len(name)-1]

        # if goal is rendered output (slow) set comboTrue to True (create a
        combo object house)
        #     this is done in following block, then we loop through rocols
        copying the combo object
        # if goal is BRAT rapid feedback (faster) set combTrue to False
        (use 3D View Texturing)
        #     3D View textured will require user input of UV mappings to
        display textures
        #     for flexibility we will bypass combo object and just copy
        individual pieces
        userIn = input("Create a combo object house? (y/n)?")
        if (newWindows == False):
            userIn = name[0:len(userIn)-1]

        if (userIn == "y"):
            combo = True
        else:
            combo = False

        combo = True

        buildHouse(name, combo)
    if (user_select == 10):
        print("you entered 10 ... selectSideFaces()")
        selectSideFaces()

```

```

def run_auto():
    dPrint("about to run_auto")
    if os.path.isdir("C:\\Blender\\runit.useUnhappy"):
        name = "unhappy"
    elif os.path.isdir("C:\\Blender\\runit.useHappy"):
        name = "happy"
    buildHouse(name, True)
    perc = 33.3
    row_cols = 9
    street_modulo = 3
    copyHouses(name, perc, row_cols, street_modulo)
    bratAnim()

##### MAIN BODY #####

two_print("brat.py: about to begin body execution ( call modeQuery() )")

#check for C:\\Blender\\runit* to determine if/how to run brat.main()
ret = modeQuery()

two_print("main body, return code from modeQuery: " + str(ret))

if (ret == "main"):
# ===== MAIN BODY CODE GOES HERE
=====
    print("this is the code for main body")
elif (ret == "merge_joint"):
    two_print("merge_joint: calling merge_joint()")
    merge_joint()
elif (ret == "auto"):
    two_print(r"found C:\\Blender\\runit_auto: so run brat.py with automatic input")
    run_auto()
elif (ret == "off"):
    two_print(r"off: not executing remaining statements (skipping main())")
else:
    two_print(r"not found 'C:\\Blender\\runit*', assuming 'quit'")

```

## REFERENCES

## REFERENCES

- [1]. Realistic Face Dynamics, Retrieved on October 24, 2011, from  
[http://people.ee.ethz.ch/~pmueller/documents/jvisp\\_special\\_issue\\_\\_pmueller.pdf](http://people.ee.ethz.ch/~pmueller/documents/jvisp_special_issue__pmueller.pdf)
- [2]. Animation Scripting Languages, Retrieved on Sept 1, 2012, from  
<http://3d.recoil.org/nojavascript/AnimTech.htm>
- [3]. Scripting Modeling Languages, Retrieved on April 26, 2012, from  
[http://bergel.eu/download/Dyla2010/dyla10\\_submission\\_6.pdf](http://bergel.eu/download/Dyla2010/dyla10_submission_6.pdf)
- [4]. API Design Matters, Retrieved on May 1, 2012, from  
<http://www.triodia.com/staff/michi/queue/APIDesign.pdf>
- [5]. Forward Kinematic Animation, Retrieved on November 29, 2012, from  
[http://en.wikipedia.org/wiki/Forward\\_kinematic\\_animation](http://en.wikipedia.org/wiki/Forward_kinematic_animation)



