Internal Workings of Acorn

Al Riddoch

March 5, 2001

Contents

1 Introduction

It has become clear from recent conversations I have had with other members of the project on irc that very little is known about how Acorn operates internally. This is not surprising as little documentation has been written on the subject, and I am relatively poor at keeping everyone else informed and discussing my work.

The aim of this document is to redress this problem by providing an up to date description of how Acorn works. It is important that this information is readily available as it can be used by client developers to develop the functionality required in their clients, and it allows the mechanisms involved to be reviewed by all WorldForge developers, but it is also important for another reason. I have now been working on Acorn for about 8 months together with many other members of the project. During this time we have developed ways of solving the problems we have come across. We have made mistakes, re-written things and changed the way worked until we got it right. We are, quite rightly I believe, proud of the achievements we have made. All this experience is wasted if the benefit of our wisdom is not passed on to future game projects.

I will assert here that the best way for future projects to develop is to build directly on what we have achieved in Acorn. Mason is set to be the key benificiary of the technology we have developed, and it is essential to the timely development of Mason as a working game that the Mason team do not waste time re-creating technologies which have already been well developed by the Acorn team.

Much of what I describe in the rest of this document is standard Atlas use, which I include because it is not well documented elsewhere. Some of it is techniques we have developed the sit on top of Atlas. Very little of Acorn is Acorn specific. We have worked hard to find the right long term solution to problems so that the same problems do not have to be solved again.

It is my hope that this document will help to bring about what I believe is one of the key aims of WorldForge. A generic interoperable system for creating games. It is my hope that any client or server which can be used with Mason will automatically work with Acorn because Acorn is a subset of Mason, In short we should be working to bring our development efforts together by using generic solutions, interoperable techniques and standards.

2 Architecture

There are two key components to Acorn, the game server, and the client. At this stage, they comprise the whole game, with one minor exception. The server is cyphesis, which in its current incarnation is written in C++ with python scripting. This version of cyphesis is referred to as cyphesis-C++ to

differentiate it from the previous incarnation which was written totally in python. The current client is uclient, a graphical isometric projection client that uses 2d images to build the scene. Each instance of the client has a single connection to a server which takes the form of a TCP connection, and Atlas is used as the communication protocol over this connection. Both the client and the server currently use the recently released stable version of the Atlas standard implementation, Atlas-C++ 0.4.0.

The exception to this architecture is an admin client used to build the world in the server at startup. This client connects to the server in the same way as a normal client, but logs in using a priveledged account called admin. Because this client is written in python, it uses the python implementation of Atlas, libAtlasPy.

3 Atlas Functionality

3.1 High Level Atlas

At the top level, Atlas is used to serialise and transmit objects across a network connection. These objects can be of two fundamental types, operations and entities. An entity type object represents any kind of entity, both in game entities such as characters, and out of game entities such as user accounts. An operation type object represents an event or message. Creation, manipulation and communication of these types is done using the Atlas::Objects api of Atlas-C++. Client server communication is achieved by sending operations between the client and server along the network connection. An operation has an argument list which usually contains one or more high level Atlas objects, i.e. entities or operations. An example of an operation with an entity argument is a move operation, sent to an in game entity to indicate that it is moving. The argument would be an entity with the id of the in game entity that is moving, and the attributes necessary to describe the movement. an example of an operation with an operation as its argument is a sight operation, sent to all in game entities when the move operation above occurs. The argument would be the move operation described above, which in turn has the entity that is moving as its argument.

3.2 Low Level Atlas and Atlas Notation

At the lower level, Atlas uses data in the form of message objects to represent the high level objects described above. A message object can be one of five types. An integer, and float, a string, a list or a mapping. A high level object is represented by a message object which is of type map, which contains various message objects keyed on strings as the operations attributes. An convention has been adopted in Atlas documentation, which will be used here, to represent low level Atlas data structure using python syntax, as it

is well suited to representing the data types involved clearly. Here are some examples of python representations of simple message objects.

42	An integer
32.85	A float
'foo'	A string
[69, 24.8, 'bar']	A list
{'name': 'foo', 'password': 'bar'}	A map representing an account

The following python represents a move operation describing a pig moving north at 1 meter per second.

```
{'objtype'  : 'op',
 'parents'  : ['move'],
 'serialno' : 12345,
 'refno'    : 0,
 'args'     : [{'objtype' : 'obj',
                'id'      : 'pig_42',
                'parents' : ['pig'],
                'loc'     : 'world_0',
                'pos'     : [0.5, 3.4, 9.7],
                'velocity': [0.0, 1.0, 0.0]}]}
```

The following python represents a sight operation of the above move operation which is sent to all entities that can see the pig move.

```
{'objtype'  : 'op',
 'parents'  : ['sight'],
 'serialno' : 12346,
 'refno'    : 12345,
 'args'     : [{'objtype'  : 'op',
                'parents'  : ['move'],
                'serialno' : 12345,
                'refno'    : 0,
                'args'     : [{'objtype' : 'obj',
                               'id'      : 'pig_42',
                               'parents' : ['pig'],
                               'loc'     : 'world_0',
                               'pos'     : [0.5, 3.4, 9.7],
                               'velocity': [0.0, 1.0, 0.0]}]}]}
```

If you think the above require more explanation, please contact the author via the worldforge irc server, or mailing lists.

3.3 Communicating with the Server

The only top level objects sent to the server should be operations. With the current version of Atlas-C++, only Atlas operations implemented by the

Atlas Objects api can be decoded by the server, so no other operations will be accepted. In Acorn other operation types may be used internally by the server, some of these may end up being sent to the client as arguments of other operations.

In order to send an operation to the server it is necessary to set the *to* attribute of the operation to indicate which server entity is performing the action. The *from* attribute can be set to the id of either an in game entity such as a character, or an out of game entity such as a player account, depending on the type of operations. The only exception to this are the operations concerned with logging into the server.

When the connection to the server is first established, there are two meaningful options. The client can send a create operation to the server in order to create a new account, or a login operation to log into an existing account. In either case the argument of the operation should be an account object with id and password atttributes as shown below.

```
{'objtype'  : 'op',
 'parents'  : ['create'],
 'serialno' : 1,
 'refno'    : 0,
 'args'     : [{'objtype'  : 'obj',
                'id'       : 'acname',
                'password' : 'pword'}]}
```

A response will be sent by the server with its *refno* set to the *serialno* of the login or create operation. The response will be an error operation if the operation failed, or an info operation with the account object as its argument in the case of success.

The account may now be used to create a character by sending a create operation with *from* set to the account, with the characters type and other details provided in an entity provided as the argument. The response will be an info operation with the completed character as its argument, with *refno* set to the *serialno* of the create operation.

Under normal circumstances all futher operations sent by the client will have *from* set to the id of the character. The operation types available are described below.

4 Modelling the World

The table below lists the attributes used to desribe in game entities in Acorn. The first section lists attributes common to all entities, the second lists attributes which are common to all entities, but which are not always set. The third section lists attributes only found on some entities, and always

present in those classes of entity, for example all characters have a drunkeness attribute. The fourth section lists attributes are sometimes set.

name	type	desription
id	string	unique id of the entity
name	string	name of the entity
loc	string	the id of the object used as reference for position
contains	list of strings	list if ids of entities which list this entity as parent
status	float	status or condition of entity
parents	list of string	lists of types associated with entity
pos	list of 3 floats	position within parent entity
velocity	list of 3 floats	velocity within parent entity
face	list of 3 floats	direction entity is oriented
drunkness	float	How drunk a character is
weight	float	weight of the entity in kilos

5 Handling Movement

Handling movement in an inteligent and practical manner is one of the areas of cyphesis that has recieved lots of attention recently. A mechanism has been developed to allow clients and NPCs to specify their movement concisely, and have the server sort out the fine details thus removing the requirement for the client to have to handle it. I will describe here the types of movement a client or NPC can perform, and how they are handled by the server. In order to perform any kind of movement, a move operation should be sent to the server, with the from attribute of the operation set to the characters id. In all cases, if the operation results in something moving that the character can observe, the server will respond with a sight of the actual move operation the happened. This will not always be the same as the move operation that was sent for reasons explained below. The client is not required to wait for the server to respond before displaying the result of the movement, but caution should be used in attempting to predict the outcome. Client side prediction is a complex subject, which will not be tackled in this document.

5.1 Turning Round

Currently in Acorn the direction a character is facing is represented by a unit vector in that direction. Normally this vector will be aligned with the horizontal plane, and behavoir if this is not the case is undefined. In order to specify that a character has changed the direction it is facing, the client should send a move operation from the character with the *face* attribute set to the new direction, as follows:

```
{'objtype'  : 'op',
 'parents'  : ['move'],
 'from'     : 'farmer_76'
 'serialno' : 0,
 'refno'    : 0,
 'args'     : [{'objtype' : 'obj',
                'id'      : 'farmer_76',
                'loc'     : 'world_0'
                'face'    : [1.0, 1.0, 0.0]}]}
```

The result should be that the *face* attribute will be set according to the operation sent, and currently there are no reasons why this should fail. There are no restriction placed on the ammount a character can turn in one operation, or the speed at which this can occur. The represention of character orientation using the *face* attribute is not Atlas compliant, and is subject to change in the near future.

5.2 Walking and Running in a given direction

The client can specify that the character is going to move in a given direction simply by sending a move operation from the character, which specifies a *velocity* vector.

```
{'objtype'  : 'op',
 'parents'  : ['move'],
 'from'     : 'farmer_76'
 'serialno' : 0,
 'refno'    : 0,
 'args'     : [{'objtype' : 'obj',
                'id'      : 'farmer_76',
                'loc'     : 'world_0'
                'velocity': [1.0, 0.0, 0.0]}]}
```

The velocity given above is 1 meter per second, which is taken to be normal walking pace. Any velocity can be specified, though this will be clipped to the maximum the player is capable of and the maximum permitted by the world.

It is assumed that the character should face in the direction in which it is moving, in which case the face attribute vector will automatically be calculated by the server. The client can specify an alternative face attribute vector if it wishes to specify that the character is not facing in the direction it is travelling, though it is currently not possible to render this kind of movement accuratly in uclient. The character in question will appear to moon-walk. A position vector should not be specified, or the operation will not be processed in the required way. The sight operation that is returned in response to this operation will include a position vector.

5.3 Walking and Running to a given destination

The client can specify that the character is going to move to a given destination simply by sending a move operation from the character, which specifies a postion vector.

```
{'objtype'  : 'op',
 'parents'  : ['move'],
 'from'     : 'farmer_76'
 'serialno' : 0,
 'refno'    : 0,
 'args'     : [{'objtype' : 'obj',
                'id'      : 'farmer_76',
                'loc'     : 'world_0'
                'pos'     : [1.0, 1.0, 0.0]}]}
```

If no velocity is given, the server assumes that the character is moving as fast as it is able, and calculates velocity as required. The direction the character is facing is handled as above for moving in a given direction. If, as is probably normally the case, the player wants their character to move at a pace other than the maximum possible speed, a velocity attribute should be provided in the direction the movement is expected to be. Clients are discouraged from offering maximum velocity as the default behavoir when moving around as it makes for an unrealistic game environment.

5.4 Moving another object

The client can specify that the character is going to move another object to a different location by sending a move operation with *from* set to the character, which specifies the require movement attributes, and the id is set to the object to be moved. This operation for example specifies that the character is picking up a pig.

```
{'objtype'  : 'op',
 'parents'  : ['move'],
 'from'     : 'farmer_76'
 'serialno' : 0,
 'refno'    : 0,
 'args'     : [{'objtype' : 'obj',
                'id'      : 'pig_82',
                'loc'     : 'farmer_76'
                'pos'     : [0.0, 0.0, 0.0]}]}
```

face and *velocity* attributes can also be specified, but it should be noted that the server will not track the movement of the object, or maintain a

record of its postion. The character will fail to perform the movement if the object to be moved is too heavy to be moved or is fixed. It should also be noted that at time of writing the server places no other restriction on this type of movement, and it is possible for a client to specify that any other object in the game is to be moved to any location. Support for this type of movement is extremely limited, but will be developed in future. Clients are discourage from allowing players to perform arbitrary movements of other objects, and should restrict moving to the type sensibly required for good gameplay.

6 Interaction with the World

6.1 Touching

The only direct means of interacting with the world used in Acorn by the client is the touch operation. This is sent by an entity to indicate that it has touched or struck another entity. Examples of its use are touch operations sent to a pig indicating that it should move, touch operations sent to a tree to make it drop acorns, or touch operations sent to a skeleton to make it collapse. The argument should be an entity with the id of the object to be touched, and to should be set to the same id.

```
{'objtype'  : 'op',
 'parents'  : ['touch'],
 'from'     : 'farmer_76'
 'to'       : 'pig_37'
 'serialno' : 0,
 'refno'    : 0,
 'args'     : [{'id'      : 'pig_37'}]}
```

6.2 Eating and Nourishment

An eat operation is sent by an entity to another whenever it is eating. The default behavoir on receiving an eat operation is to ignore it, but if the object is edible, then it will return a nourish operation to the sender indibating how much nourishment has been gained. The first argument of the eat operation is the object to be eaten, and the second is the entity that is eating it. The second argument is required to make sure that the correct nourish operation can be sent. The argument of the nourish operation is an entity with the id of the object to be eaten, and a weight attribute with a value representing the nutrition gained. This value is by default the same as the weight of the object being eaten, but is not always this value. For balance and realism some objects have an effective nutrition weight which is higher or lower than their actual weight.

```
{'objtype'  : 'op',
 'parents'  : ['eat'],
 'from'     : 'pig_36'
 'to'       : 'acorn_95'
 'serialno' : 0,
 'refno'    : 0,
 'args'     : [{'id'      : 'acorn_95'},
               {'id'      : 'pig_36'}]}

{'objtype'  : 'op',
 'parents'  : ['nourish'],
 'from'     : 'acorn_95'
 'to'       : 'pig_36'
 'serialno' : 0,
 'refno'    : 0,
 'args'     : [{'id'      : 'acorn_95',
                'weight'  : 0.1}]}
```

7 Perception

When a character is created in the world, or a new connection is established to an existing character, nothing is initially known about the world surrounding that character. In order to query the world, it is necessary to send look operations to all the surrounding entities in order to receive sight operations which describe those entities. In order to bootstrap this process, the client should send a look operation with *from* set to the character, *to* unset, and no argument. The server will route this operation to the highest level entity in the world that the character is able to see. In the case of Acorn, the highest level entity is always the world, which is a special entity which has id 'world_0'.

```
{'objtype'  : 'op',
 'parents'  : ['look'],
 'from'     : 'farmer_76'
 'serialno' : 0,
 'refno'    : 0}
```

The server will respond with a sight operation with the world as its argument. The important attribute of the world entity is the contains attriubute, which is a list of the ids of all the entities in the server which use the world as a reference for position that the character can see. This list should then be used to send further look operations to these entities in order to build up a full picture of the world.

It is important that the client is able to differentiate between the kind of sight operation sent in response to a look operation, which has an entity as its argument, and a sight operation which sent to notify the client of an event that has happened in the world, which has an operation as its argument. This second type of sight operation will have the operation as it occured in the world. In Acorn 0.3 every character is able to witness every event that happens in the world, so no change can occur to an entity without the client being aware of it. In future versions of Acorn, and later games, there will be many reasons why events happen in such a way that a sight operation does not get sent to every client. The most immediate reason will be that the event occured too far away for the character to be able to see it, but as the complexity of games increases other reasons such as environmental conditions, characters abilities, and line of sight will be implemented.

Many of the sight operations will be of operations listed in sections 3.3, 5 and 6, but sight operations will also be received for operations that occur internally to the server, most of which are standard Atlas operations, some of which are custom operations.

7.1 Standard Operations

7.1.1 Set

A sight of a set operation will be observed when an attribute of an entity has been changed. The argument will be an entity with the id of the entity affected, and the changed attributes with their new values.

7.1.2 Delete

A sight of a delete operation will be observed when an entity is destroyed. The argument of the operation will usually be the entity that has been destroyed, but all that is actually required is the id of that entity.

7.2 Custom Operations

7.2.1 Fire

A sight of a fire operation will be observer when an entity is on fire. The argument of the operation will be an entity with the id of the fire entity, and a *status* attribute. The *status* attribute represent both the status and the size of the fire as they are synonymous. A *status* attribute of 1.0 indicates a fire about the size expected in a small campfire.

8 Modelling Systems

Various systems have to be modelled in order to create a working virtual world that is the foundation of Acorn. In all cases these systems are modelled using attributes of entities, which are generally either numerical or string types, and the exchange of operations between entities.

8.1 Biological Systems

8.1.1 Plants

The only important plants currently modelled in Acorn are oak trees. For the most part the trees are modelled using a simplified version of of the generic tree code. Each tree has a *fruits* attribute which is the number of fruits, or acorns in this case, which are still in the tree. The tree gets a tick operation every minute, and when this happens a random number of fruits between zero and two fall from the tree, and then a new fruit is added to the tree. Until the fruit falls from the tree, it does not actually exist as an entity in the world, so a create operation is used to create the fruit that falls, with the position set to a random point on the ground under the tree.

The only difference in behavoir between an oak tree and the generic tree is that an oak tree will drop a random number of fruits between zero and two if it receives a touch operation.

8.1.2 Animals

Pigs, wolves and crabs are all modelled using the same animal code for the purposes of nutrition and growth. Their concious behavoir is modelled using the same AI system used for non-player characters as detailed in section 9. The animal code deals with how hungry an animal is, how quickly it processes food it has eaten, under what circumstances it will put on weight, or lose it, and in extreme cases when it will starve to death. The *status* and *weight* attributes are the two most important attributes in modelling the above, but there is also a third private attribute called *food*, which is the ammount of undigested food the animal has in its stomach. This attribute is not available publicly, and is not controlled using set operations, but it is necessary to keep the animals mind informed of this attribute so that its behavoir can depend on its appetite. For this reason sight of set operations are faked and sent to the animals own mind with the value of the *food* attribute.

8.1.3 Undead

There are two types of undead entity present in Acorn, though neither currently have a very strong role. The first is the skeleton and the other is the

lych.

The skeleton has a very simple mind which does not support the advanced features such as language and goals used by characters, but has an instinct to move towards any object it sees moving. It does this by sending a move operation whenever a move operation is recieved. In order to prevent operation loops, and weird behavoir the skeleton only responds to move operations by a small number of entity types. The skeletons body likewise is modelled simply. Whenever the skeleton entity receives a touch operation, it disintegrates, as skeletons are not very tough. This disintegration is modelled by a number of operations. Firstly a number of create operations are sent to create one entity each of the types skull, ribcage, pelvis, thigh, shin and arm. These objects are created scattered over a small area surrounding the skeleton. Secondly a set operation sets the *status* of the skeleton entity to zero, meaning that the entity get deleted, and gives the impression that the entity has disintegrated leaving behind its component parts.

The lych is currently invisible, but as the clients ability to handle variable transparency is improved it will become as translucent figure. The lych also does not work correctly yet. The lych's purpose is to keep the skeleton population around by reassembling skeletons from the bones that are left around the map. It creates a new skeleton by moving the required bones into a pile, sending set operations to delete them all, and a create operation to create the skeleton entity.

8.2 Nonbiological Systems

8.2.1 Fire

A fire entity is located with reference to the entity that is on fire, so is always listed in that entity's contains attribute. A fire entity gets a tick operation every thirty seconds, and each time this happens it sends a fire operation to its entity, and a sight of that fire operation is created. If the entity is at all flammable, it should calculate the proportion of its weight that has burned and reduce its *status* attribute accordingly (using a set operation). The proportion of the objects weight that has burnt will depend on the objects *burn_speed* attribute, and the *status* of the fire entity, which will be given as an attribute of the argument of the fire operation. The weight of the entity consumed by the fire should then be returned to the fire entity using a nourish operation, in exactly the same way as used to handle eating as described in section 6.2. The numerical values used in this calculation have been carefully balanced to allow realistic fires to happen in Acorn. The *burn_speed* attribute should be estimated as the proportion of a given object that would be consumed in one thirty secend period by a fire with a status of 1.0. The Lumber entity which represent a log with a weight of 10kg has a *burn_speed* of 0.01 which means that it would take a fire with a *status*

of 1.0 50 minutes to consume it. This value, in combination with carefully balanced factors in the calculation, also means that if the entity was ignited with a very small fire, it would not burn quickly enough to nourish it into a larger fire, so the fire would go out. In order to make a small fire grow into a large one, it is necessary to find faster burning material such as straw or kindling. Such material might have a *burn_speed* between 0.2 and 0.5. A *burn_speed* of 1.0 indicates an extremely fast burning material, and values of more than 1.0 represent materials that burn with an almost explosive speed.

8.2.2 Weather

It was not clear initially how to correctly model weather in the world, so an initial very simple attempt was made. The world contains an entity of type weather which has one functioning attribute called *rain*. This attribute when set to 1.0 indicates that rain is falling, and when set to 0.0 indicates that no rain is falling. This area will be subject to much development, and will include a *snow* attribute, and varying values betwwen 0 and 1 indicatint different intensities of weather.

9 Artificial Intelligence

This section is only an overview of the capabilities of the AI system used in Acorn. For more detailed information access to the cyphesis documentation will be required, and Aloril is usually available on the WorldForge irc server at irc.worldforge.org.

9.1 language

Characters understand language using the interlinuish code. Language is recieved in the form of text as the say argument of a talk operation. The talk operation will itself be the argument of a sound operation. The text is analysed and converted into a structured form, though currently only a small number of sentence constructions, and a limited vocabulary of nouns and verbs are understood. It is important to note that if a type of entity is to be understood by NPCs in speech, the noun representing that entity type must be added to the interlinuish code.

The main use of the interlinguish code when defining NPC behavoir is through the use of trigger functions. Trigger functions can be defined in the mind of an entity, according to a naming convention which means that the trigger function is automatically called when a sentence of the form of interest is decoded, without the need to register the function. These same trigger naming conventions can be used when creating trigger goals, otherwise known as dynamic goals, which will be mentioned in section 9.3.3.

9.2 memory

Characters have a memory which is used to store information about all entities in the world which that character has received perception operations from. Every time a character received a perception operation, the data contained is used to update the information about the world in the memory data structures. This memory is the foundation on which all high level mind code is based.

9.3 goals

A goal has two essential features. It has a conditional, and a list of sub goals. This list of sub goals can contain references to goal objects, or to functions. The conditional is a value or function which indicates whether the goal has been completed. Each time the goal is polled, this is checked, and if it is not true, then the sub goal list is traversed until a function is found, and executed. Once a function has been called and has return an action, nothing more is done on this goal until the next tick. The list of sub goals contains references to goals, or functions each of which must be completed or executed in order.

Each NPC has a top level list of goals which are evaluated in order of their importance. If the most important goal is not complete, then the character will devote all its attention to this goal. Each atomic goal, by which I mean a goal without any sub goals, is simple in that it defines a very simple condition, and performs a simple action, but complex goals can be constructed easily by building a hierarchy of goals using these simple atomic goals.

9.3.1 Instinct Goals

Animals have instincts, which are represented by goals. Most of these goals are simpler that the goals used by NPCs, but NPCs also have instincts as well as complex goals. Goals used by animals are typically of the for finding food, hunting prey or avoiding an enemy or predator.

9.3.2 Sentient Goals

Characters have higher level goals, which can be used to model quite complex behavoir. Examples of such goals are a goal to sell pigs at the market place, or in the case of the butcher, buy pigs in the market place and use a cleaver to chop the pigs up into ham. Care has been taken to keep the goals extremely generic by parameterising the types of entity, and action involved. The goal used by the pig merchant could in fact be used to sell any good which has a fixed unit price, and the goal used by the butcher could be used to perform

any trade or craft which involves purchasing entities of one type at a cost based on weight, using a tool on those entities, and selling the result.

9.3.3 Trigger Goals

Whereas the goals mentioned above are always active, and this suitable for continuous or long term activity, there is a need for actions to be taken under certain circumstances, without the overhead of checking all the time whether the circumstances are right. Trigger goals are used in this case. Each trigger goal has a string which represents the type of event which might trigger it. The mind code which handles the trigger subsystem will activate the goal when an event that matches the trigger string occurs. The trigger goal can then perform an action, or activate some more complex code to deal with the circumstances.

By way of example, the butcher has a trigger goal which is activated whenever a character expresses the desire to sell a pig. The action performed by this goal when activated is to calculate the value of the pig based on its weight, and the butchers knowledge about the value of pigs, and calculate whether the butcher has enough money. If so, transfer the money to the vendor, and add the pig to this characters inventory. All this happens instantaneously.

An example of more complex code is the extinguish fire goal, which is not currently used as it needs tuning, and is not required for Acorn. When a sight of a fire operation is received, the trigger goal is activated. All this trigger goal does is add an ordinary goal to the characters goal list at top priority. The normal goal deals with extinguishing the fire, and once this is done it signals that it should be removed by setting its irrelevant attribute, allowing the character to proceed with normal business.

10 Installation Layout

This section is mostly concerned with the directory layout under Linux and other unix-like operating systems. Directory layout follows where possible conventions used in modern Linux distributions as defined by the Linux Standard Base (LSB). Most WorldForge components are built using build files generated using the GNU autoconf family of tools. These tools allow a prefix to be specified for the installation of those components. For convenience the binary distributions of Acorn have been build with the prefix set to /opt/forge/ so copies can easily be installed, moved around and compared. Binary packages have been built with the standard prefix, /usr. In accordance with standard unix-like convention, binaries are stored in ¡prefix¿/bin and libraries in ¡prefix¿/lib.

10.1 Program specific data files

Acorn, and other WorldForge games, require a large quantity of platform neutral data in the form of images, sound and music, which collectively are referred to as media. Care has been taken to identify media files that are used or potentially used by more than one WorldForge component, and keep them separate from those files which are only used by one component. Program or component specific media files are kept in ¡prefix¿/share/¡component¿/ where component is the name of the WorldForge component the files belong to. If any platform specfic data files are associated with a component, they should be stored in ¡prefix¿/lib/¡component¿/, though no such files are found in Acorn. In accordance with LSB, component specfic documentation such as readme files and copyright notices, as well as more complex documentation, should be installed in ¡prefix¿/share/doc/¡component¿-¡version¿/.

10.2 Shared data files

Platform neutral shared data files that are used or potentially used by more than one WorldForge component are kept in ¡prefix¿/share/forge. Within this directory a preliminary standard has been initiated to organise files cleanly so that all WorldForge components can access them easily. Files from the WorldForge cvs modules called media should be kept in ¡prefix¿/share/forge/media/, with the same pathname is they have within the CVS modules. This allows WorldForge developers to place a symbolic link in ¡prefix¿/share/forge/ to a copy of the media module to remove the need for media to be specially installed. The media directory should contain a symlink to the media-2d module which now contains the majority of new 2d media. A script has been written, and can be found in the uclient source directory, which searches through the uclient source for references to media files in the media repository and creates a list of files which can be used to create a media package for distribution. This script, called get_media_files.pl, is used to automatically build the acorn-media package for distribution, as detailed in section 11.

11 Buidling Acorn

The WorldForge components used in this version of Acorn are uclient, cyphesis-C++, Atlas-C++ and varconf.

11.1 Automated Distribution

In order to ensure that the build that the build of Acorn binary release package is totally repeatable, Makefiles have been written to automate the process. These Makefiles contain a number of rules which build each part of the distribution.

acorn_media builds a media package by extracting the filenames of the required media from the uclient source using a perl script, and using the list produced to build a tar file.

acorn builds binary packages by building and installing the code into /opt/forge, and creating tar files of the result, including media in some cases. In order for this rules to work, it depends on the acorn_media rule, and relies on uclient media packages being placed in /opt/acorn/media/.

acorn_dist build source distributions for all WorldForge components used in Acorn, using standard autoconf rules where possible.

acorn_rpm is intended to build binary RPM packages for all WorldForge components used in Acorn, but has not been completed to my satisfaction yet.

all the default rule, calls the acorn_dist and acorn rules, and thus implicitly the acorn_media rule, in order to create a complete distribution of Acorn.