MicroCosm™ Object Manual

the MicroCosm Object Set described in excruciating detail

Chip Morningstar
Lucasfilm Ltd. Games Division
April 1, 1986

Introduction

This document describes the basic set of objects from which the **MicroCosm** fantasy world is constructed. It both augments and*super sedes* the earlier document *MicroCosm Minimal Object Set*. This document describes what the various objects are and specifies them in sufficient detail that they may actually be programmed.

The Object Environment

Each object is a member of some *class* of objects, i.e., it is one of a group of objects that are all of the same basic type. An object's class determines what its behavior is going to be and how it is going to look on the screen. Each class is represented in the **MicroCosm** system by a *class descriptor*. The class descriptor is a collection of data and executable code that defines objects of a given type. Creating the object set really means creating the class descriptors for all of the classes of objects described in this document.

Any actual object is an *instance* of a particular class. Each object is represented by an *instance descriptor*. Instance descriptors are created and destroyed as objects come and go from the system. Each instance descriptor is simply a small area of memory set aside to contain the data that describes the state of a particular object (as distinguished from other objects of the same class). The memory address of an object's instance descriptor can serve, internally, as a unique identifier for the object itself.

This abstract model of classes and instances is used both in the home and in the host systems. The format and content of the class and instance descriptors will differ across systems, of course, depending on the role that each half of the system plays in the object's existence.

Class Descriptors

The information describing a class consists two things: *imagery* and *behavior*. The class descriptor itself is just a data structure that points at these things. The actual imagery and behavior are stored separately. This way, classes can share imagery and behavior when they overlap between classes (which, in the case of behavior, is common).

Imagery consists of the animation cels or graphics driver tables required for the graphics software to display the object. Most objects consist of a single static image. A few have a small number of images, one of which is displayed at any given time depending on the state of the object. Some have a more complex set of images that are used to vary the appearance of the object according to some parameter (so that all objects of the class will not appear identical on the screen) or to animate the behavior of the object (avatars are the primary example of this). In any case, the imagery is a set of static data that is used by the graphics routines. Interpretation of this data may depend on object state information that the graphics routines obtain from the object's instance descriptor.

Behavior consists of a set of executable routines that are called by the object database software. The class descriptor contains a vector of entry points. Each element in this vector has a particular, pre-assigned meaning. Each entry point corresponds to a particular function that the object can be called upon to perform. Some of these functions are standard and all classes can perform them. These include responding to any of the verbs used in the player interface. Other functions are specific to the class itself and will be called as the result of messages from the opposite end of the communications line (the host if this is the

home system, and vice-versa).

The class descriptor in the home system is laid out as follows:

Byte	*Entry*	*Description*
0–1	**length**	Length of the class descriptor
2	**image#**	Number of images this class uses
3	**sound#**	Number of sounds this class uses
4	**action#**	Number of behaviors this class uses
5	**imageOff**	Offset in this class descriptor to first image ptr
6	**soundOff**	Offset in this class descriptor to first sound ptr
7	**actionOff**	Offset in this class descriptor to first behavior ptr
8	**objSize**	Length to create objects of this class
9	**initSize**	Number of initialization bytes expected from the host
10	**capacity**	Number of objects containers of this class can hold
11	**maxOccp**	Number of occupants seats of this class can hold
12–N-1	*images*	Image pointers for this class
N–M-1	*sounds*	Sound pointers for this class
M	**do**	Action for the standard verb **do**
M+1	**rdo**	Action for the standard verb **reversed do**
M+2	**go**	Action for the standard verb **go**
M+3	**stop**	Action for the standard verb **stop**
M+4	**get**	Action for the standard verb **get**
M+5	**put**	Action for the standard verb **put**
M+6	**talk**	Action for the standard verb **talk**
M+7	**destroy**	Destroy an instance of this class
M+8+	**...**	Asynchronous actions directed by the host

There may be additional entry points beyond this. These correspond to the class-specific behavior functions discussed above.

Instance Descriptors

An instance descriptor is a data structure that describes the state of one particular object. Each instance descriptor is a simple vector of bytes. This vector has two parts: the first ten bytes form the *generic part* and the remaining bytes (if any) form the *class-specific part*. The generic part contains information that is common to all objects, regardless of class. The layout of the generic part of the instance descriptor is the same for all objects, so that external software can manipulate the information it contains without having to know what sort of object it is part of. The class-specific part contains information that depends on what sort of object the object is. The interpretation of this information is solely at the discretion of the behavior code associated with the particular class of object.

The instance descriptor in the home system is laid out as follows:

Byte	*Entry*	*Description*
0-1	**length**	Length of this instance descriptor
2	**noid**	Identifier by which this object is known to host
3	**class**	Class of this object
4	**grstyle**	Which style of this class of object to display as
5	**x**	X-coordinate position within containing object
6	**y**	Y-coordinate position within containing object
7	**orient**	Orientation to display object in
8	**grstate**	Graphic state to display object in
9	**aniStart**	Starting graphic state of animation sequence
10	**aniEnd**	Ending graphic state of animation sequence
11	**container**	Object containing this object
12-N	**contents**	Contents of container, if any
N+	**...**	Class-specific information

The structure of an instance descriptor in the host system depends on the structure of the host database, which has not yet been defined. However, the generic information it contains will be the same except for graphic state data (which is obviously not needed in the host). In the object descriptions that follow, we will speak as if the host instance descriptor was essentially the same as the home instance descriptor, though in actual practice the access methods used will differ. Class-specific information in the host will depend on the needs of the class' behavior code, as it does in the home system.

The Object Descriptions

What follows is a series of object descriptions, one for each class of objects in the basic object set. Each description has the following form:

Object:
name of object
Description:
A brief description of the object.
Function:
The object's purpose in the MicroCosm world.
Notes:
Anything else that needs to be said about the object.
Styles:
Stylistic variations possible for this object.
Properties:
The object's properties, i.e., the contents of the class-specific part of the home system instance descriptor.
Class properties:
Properties associated with the object's class.
Host properties:
Additional object properties that are kept only in the host.
Command Behavior:
Do:
The action to take for the verb **do**.
Go:
The action to take for the verb **go**.
Stop:
The action to take for the verb **stop**.
Get:
The action to take for the verb **get**.
Put:
The action to take for the verb **put**.
Talk:
The action to take for the verb **talk**.
Reversed Do:
The action to take for the verb **reversed do**.
Initialization:
The action to take on object initialization.
Destruction:
The action to take on object destruction.
Asynchronous actions:
Other behavior that the object should be capable of as a result of asynchronous messages from the host or calls from other objects or other parts of the home system.
Graphics:
The graphics required of the object.
Host behavior:
Behavior functions resident in the host.

Throughout the descriptions we use a **C**-like pseudo-code to describe specific behaviors. A few words about the notation are appropriate:

Object properties (the contents of an instance descriptor), both generic and class-specific, are referred to by name using identifiers in `this typewriter-like typeface`. Such references look like **C**

struct references of the form `object.property`. The identifier`self` refers to the object whose beha v-ior we are executing, so that, for example `self.class` is the object's class number and `self.x` is the object's X-coordinate location. Similarly, the identifier `avatar` refers to the instance descriptor of the player's avatar, `region` refers to the instance descriptor of the current region, and `selfClass` refers to the class descriptor of the object `self`. Identifiers in`CAPITALS LIKE THIS` are symbolic constants which should be described in the text.

One peculiar expression you will see frequently in the home system object behaviors is:

```
@ object!ACTION (arg1, arg2, ...) → (result1, result2, ...)
```

This is an instruction to send a message to the host. `object` denotes the object to whom the message is addressed. `ACTION` denotes the particular action from that object that is desired. The `arg`s are any arguments to be sent with the message. The `result`s are the identifiers to bind to the successive bytes of the response. A similar form:

```
ACTION (arg1, arg2, ...) → (result1, result2, ...)
```

is used in the host system behavior definitions as a header for a single behavior definition, much like a **C** function header.

Some other odd expressions are found in the host system object behaviors:

```
object → ACTION* (arg1, arg2, ...)
```

This is an instruction to send a message back to the home system. `object` denotes the object to whom the message is addressed. `ACTION*` denotes the particular action the object should tak e. The`arg`s are an y arguments to be sent with the message. Note that there is only a single host system copy of any particular object, since there is only one host. However, there are as many home system copies of an object as there are avatar's in the current region. This expression indicates that the message indicated should be sent to the computer of the player whose behavior request we are currently processing. The expression:

```
# object → ACTION* (arg1, arg2, ...)
```

is similar, except that the message is to be sent to every player in the region *except* the one whose behavior request we are currently processing.

Also, in messages to the host, the home system refers to objects using a single byte object identifier. The expression

```
↑ objectId
```

denotes the object instance descriptor for the object to which `objectId` refers.

Flow-of-control constructs, function and procedure calls, expressions, and other such things resemble those found in **C**.

Home System Standard Actions and Functions

In the object descriptions that follow, these functions are often used for or within object behaviors. These standard functions are provided because the objects share a common underlying world model and there is much redundancy in the user interface. They provide both consistency and a significant reduction in the amount of code which must be created and stored.

```
adjacent(object)
{
    /* Test for adjacency. To be adjacent, object must be within a foot of
   the avatar but not right where the avatar is. */
    return(object.container==region && (abs(object.x - avatar.x)==1 ||
                                        abs(object.y - avatar.y)==1))
}

adjacentOpenClose()
{
    /* If adjacent, open/close door or gate.  Otherwise, depends. */
    if (adjacent(self)) {
        openClose()
    } else {
        depends()
    }
}

adjacentOpenCloseContainer()
{
    /* If adjacent, open/close container.  Otherwise, depends. */
    if (adjacent(self)) {
        openCloseContainer()
    } else {
        depends()
    }
}

amongContents(object, container)
{
    for (i=0; i<(lookupClass(container.class).capacity; ++i) {
        if (container.contents[i] == object) {
            return(TRUE)
        }
    }
    return(FALSE)
}

answer(phone)
{
    /* Answer the phone if it is ringing. */
    @ phone!ANSWER () → (success)
    soundEffect(SOUND_CLICK_CLUNK)
    avatar.action = UNHOOK
    if (success) {
        phone.state = PHONE_TALKING
        soundEffect(SOUND_SILENCE)
    } else {
        soundEffect(SOUND_LINE_THUMP)
```

```
        if (phone.class == PHONE_BOOTH_CLASS) {
            phone.state = PHONE_OFF_HOOK
        } else {
            phone.state = PHONE_ACTIVE
        }
        soundEffect(SOUND_DIAL_TONE)
    }
}

answerOrUnhook(okToProceed, unhookedState)
{
    /* Answer the phone if it's ringing, otherwise just pick up the
    receiver. */
    if (okToProceed) {
        if (self.state == PHONE_RINGING) {
            answer(self)
        } else if (self.state == PHONE_READY) {
            @ self!UNHOOK () → ()
            self.state = unhookedState
            avatar.action = UNHOOK
            soundEffect(SOUND_CLICK_CLUNK)
            soundEffect(SOUND_DIAL_TONE)
        }
    }
}

balloonMessage(source, format, arg1, arg2, arg3)
{
    /* Output text using the Balloon-O-Matic. source denotes
    the object that is "speaking", i.e., the object to which the quip
    should point. format and the args specify the text
    to print using the conventions of the Unix printf function. */
    /* In Randy's Balloon-O-Matic code. */
}

broadcast()
{
    /* Broadcast typed text to everyone in region */
    balloonMessage(avatar, text)
    @ avatar!SPEAK (NULL, text) → ()
}

cease()
{
    /* Stop any on-going action. */
    /* *** nop? *** */
}

changeContainers(object, newContainer, newX, newY)
{
    /* Move object from wherever it is now to the indicated
    (x, y) location in the indicated new container. */
    object.container.contents[object.y] = NULL
    object.container = newContainer
```

```
    object.x = newX
    object.y = newY
    newContainer.contents[newY] = object
}

climbInOrOut()
{
    /* If in a vehicle, get out.  If adjacent, get in.  If elsewhere, walk
    to it.  Only works when vehicle is not moving. */
    if (!self.moving) {
        if (elsewhere(self)) {
            goTo()
        } else {
            sitOrGetUp()
        }
    }
}

concatenateStrings(s1, s2)
{
    /* *** Return the concatenation of the two strings s1 and
    s2. *** */
}

createObject(class, container, x, y)
{
    /* Create and object of the specified class located at the given
    (x,y) location within the given container.  Return a pointer to the new
    instance descriptor. */
    /* In Ron's database code. */
}

createPaperObject(text)
{
    /* Create a paper object with text on it. */
    object = createObject(PAPER_CLASS, region, avatar.x, avatar.y)
    object.text = text
    return(object)
}

damageAvatar(who, damage)
{
    /* When an avatar is injured, subtract damage from his health
    and apply the appropriate effects accordingly. *** figure out what this
    means and how to do it. *** */
}

depends()
{
    /* Punt to reversed do. */
    target = self
    if (!emptyHanded()) {
        self = avatar.inHand
    } else {
```

```
        self = avatar
    }
    (self.reversedDo)()
}

destroyObject(object)
{
    /* Destroy the given object.  I.e., delete it and garbage collect
    all its resources. */
    /* In Ron's database code. */
}

destroyObjectDramatically(object)
{
    /* When we want to blow something up, rather than simply deleting
    it. */
    (object.destroy)()
    destroyObject(object)
}

dialOrTalk(okToProceed)
{
    /* If ok to proceed, interpret text as a phone call: first text message
    is phone number, further messages are conversation with the person at
    the other end.  When you enter the number, it is dialed.  If there is
    an answer, you can talk.  If not, the phone is deactivated when you
    hang up.  If not active, broadcast. */
    if (okToProceed && self.state == PHONE_ACTIVE) {
        @ self!DIAL (text) → (success)
        soundEffect(SOUND_DIAL, text)
        if (success) {
            self.state = PHONE_LINE_RING
            soundEffect(SOUND_LINE_RING)
        } else {
            self.state = PHONE_LINE_BUSY
            soundEffect(SOUND_BUSY_SIGNAL)
        }
    } else if (okToProceed && self.state == PHONE_TALKING) {
        balloonMessage(avatar, text)
        @ self!TALK (text) → ()
    } else {
        broadcast()
    }
}

doMagic()
{
    /* If in hand, do magical function.  Otherwise, depends. */
    if (holding(self)) {
        @ self!MAGIC () → ()    /* Works asynchronously. */
    } else {
        depends()
    }
}
```

```
doMagicIfMagic()
{
    /* If magic and in hand, do magical function.  Otherwise, depends. */
    if (holding(self)) {
        if (self.magic) {
            @ self!MAGIC () → ()    /* Works asynchronously. */
        }
    } else {
        depends()
    }
}

dropAt()
{
    /* Drop the object in hand on the ground at the location of self. */
    putInto(region, self.x, self.y, DROP)
}

dropHere()
{
    /* Drop the object at the avatar's location. */
    putInto(region, avatar.x, avatar.y, DROP);
}

dropInto()
{
    /* Drop the object in hand in the container self, if you can. */
    putInto(self, NULL, NULL, PUT)
}

dropIntoIfOpen()
{
    /* Drop the object in hand in the container self, if it's open. */
    if (self.open) {
        putInto(self, NULL, NULL, PUT)
    } else {
        putInto(region, self.x, self.y, DROP)
    }
}

dropOnto()
{
    /* Drop the object in hand on self, if you can. */
    putInto(self, NULL, NULL, PUT)
}

elsewhere(object)
{
    /* Test if object is elsewhere.  To be elsewhere, object must be
   neither where the avatar is nor adjacent to him nor on his person. */
    return(!here(object) && !adjacent(object) && object.container!=avatar)
}

emptyHanded(who)
```

```
{
    /* Return TRUE iff the avatar is not holding anything. */
    return(who.inHand == NULL)
}

enterOrExit()
{
    /* If not adjacent to self, go to it.  If adjacent, enter it.  If
    inside, exit it. */
    if (here(self)) {
        goExit()
    } else {
        goOnto()
    }
}

explosionAt(x, y, size)
{
    /* *** Animate an explosion of the given size at the given
    location. *** */
}

get(object)
{
    /* Tell host to grab the object, then update location and containership
    if the operation was successful. */
    if (emptyHanded(avatar)) {
        @ object!GET () → (success)
        if (success) {
            changeContainers(object, avatar, NULL, HAND)
            avatar.action = PICKUP  /* *** What if picking from container? *** */
            subjectObject = object
            return(TRUE)
        }
    }
    return(FALSE)
}

goAcross()
{
    /* Walk to position opposite from present position w.r.t. the object's
    location, if you can. */
    if (here(self)) {
        return(goPreferred())
    } else if (adjacent(self)) {
        return(goXY(oppositeX(self.x), oppositeY(self.y)))
    } else {
        return(goTo())
    }
}

goEnter()
{
    /* Walk into object, via preferred adjacent position. */
```

```
    return(goTo() && goPreferred() && goOnto())
}

goExit()
{
    /* Walk out of object to preferred adjacent position. */
    return(goPreferred())
}

goOnto()
{
    /* Walk to the object's location, if you can. */
    return(goXY(self.x, self.y))
}

goOntoAndDropAt()
{
    /* Put hand-held item at object's location. */
    if (goOnto()) {
        dropAt()
    }
}

goPreferred()
{
    /* Walk to preferred adjacent position. */
    goXY(self.preferredX, self.preferredY)
}

goTo()
{
    /* Walk to nearest position adjacent to the object's location, if you
    can. */
    return(goXY(nearestAdjacentX(self.x), nearestAdjacentY(self.y)))
}

goToAndDropAt()
{
    /* Put hand-held item by object's location. */
    if (holding(self)) {
        dropHere()
    } else if (goTo()) {
        dropAt()
    }
}

goToAndDropInto()
{
    /* Put hand-held item into object. */
    if (holding(self)) {
        dropHere()
    } else if (goTo()) {
        dropInto()
    }
```

```
}

goToAndDropIntoIfOpen()
{
    /* Put hand-held item into object if it's open. */
    if (holding(self)) {
        dropHere()
    } else if (goTo()) {
        dropIntoIfOpen()
    }
}

goToAndDropOnto()
{
    /* Put hand-held item onto object. */
    if (holding(self)) {
        dropHere()
    } else if (goTo()) {
        dropOnto()
    }
}

goToAndFill()
{
    /* If the object in the avatar's hand is an empty container, tell the
    host to fill it up and update our state info accordingly. */
    if (goTo()) {
        if (isEmptyContainer(avatar.inHand)) {
            @ avatar.inHand!FILL () → (success)
            if (success) {
                self.filled = TRUE
                avatar.action = FILL
            }
        }
    }
}

goToAndGet()
{
    /* Pick object up, put in hand. */
    if (holding(self)) {
        return(TRUE)
    } else if (goTo()) {
        return(get(self))
    } else {
        return(FALSE)
    }
}

goToAndPickFrom()
{
    /* General routine to pick stuff up out of containers */
    if (goTo()) {
        get(pickObject(self.contents))
```

```
    }
}

goToAndPickFromIfOpen()
{
    /* General routine to pick stuff up out of open/closable containers. */
    if (goTo()) {
        if (self.open) {
            get(pickObject(self.contents))
        }
    }
}

goToAndPickFromOrGet()
{
    if (!holding(self)) {
        if (goTo()) {
            if (self.open) {
                get(pickObject(self.contents))
            }
        }
    } else {
        get(self)
    }
}

goToCursor()
{
    /* Go to location designated by the cursor. */
    goXY(cursorX, cursorY)
}

goToCursorAndDropAt()
{
    /* Put hand-held item at cursor location. */
    if (goToCursor()) {
        dropAt()
    }
}

goToOrPassThrough()
{
    /* If adjacent, go through (opening/closing as needed), else goto. */
    haveKey = (holdingClass(KEY_CLASS) && avatar.inHand.keyNumber == self.key)
    if (adjacent(self)) {
        if (self.open || self.unlocked || haveKey) {
            return(goXY(oppositeX(self.x), oppositeY(self.y)))
        } else {
            return(FALSE)
    } else {
        return(goTo())
    }
}
```

```
goXY(x, y)
{
    /* Walk to a particular (x, y) location, if you can. */
    @ avatar!WALK (x, y) → (destinationX, destinationY)
    avatar.x = destinationX
    avatar.y = destinationY
    /* If we couldn't walk anywhere, the destination coordinates will be
   the same as our current location, and the WALKING activity will
   terminate immediately */
    avatar.activity = WALKING
    return (destinationX == x && destinationY == y)
}

hangUp()
{
    /* Hang up the phone. */
    soundEffect(SOUND_THUMP)
    @ self!HANG () → ()
    self.state = PHONE_READY
    soundEffect(SOUND_SILENCE)
    avatar.action = HANGUP
}

hangUpOrAnswer(okToProceed)
{
    /* If test passed, hang up phone if not already hung up.  Answer if
   ringing.  Otherwise, depends. */
    if (okToProceed) {
        if (self.state == PHONE_OFF_HOOK || self.state == PHONE_TALKING ||
                self.state == PHONE_LINE_BUSY || self.state ==
                PHONE_LINE_RING) {
            if ((self.state == PHONE_LINE_BUSY || self.state ==
                    PHONE_LINE_RING) && self.class == PHONE_BOOTH_CLASS) {
                soundEffect(SOUND_JANGLE)
                refund(PHONE_CALL_COST, avatar)
            }
            hangUp()
        } else if (self.state == PHONE_RINGING) {
            answer(self)
        }
    } else {
        depends()
    }
}

here(object)
{
    /* Test if object is here.  To be here, object must be where the avatar
   is but not on the avatar's person. */
    return(object.container == region && object.x == avatar.x &&
        object.y == avatar.y)
}

holding(object)
```

```
{
    /* Return TRUE iff the avatar is holding the given object. */
    return(avatar.inHand == object)
}

holdingClass(class)
{
    /* Return TRUE iff the avatar is holding an object of class class */
    return(!emptyHanded(avatar) && avatar.inHand.class == class)
}

load(object)
{
    /* Compute the load inside object, i.e., how many things it
   is holding. */
    result = 0
    for (i=0; i<(lookupClass(object.class).capacity; ++i) {
        if (object.contents[i] != NULL) {
            result++
        }
    }
    return(result);
}

lookupClass(classNumber)
{
    /* Return the class descriptor associated with the given class
   number. */
    /* In Ron's database code. */
}

mediumExplosion()
{
    /* When items of major magic or explosive content are destroyed. */
    explosionAt(self.x, self.y, MEDIUM_EXPLOSION)
}

nearestAdjacentX(x)
{
    /* Return X coordinate of nearest adjacent location to object. */
    if (avatar.x == x) {
        return(x)
    } else if (avatar.x < x) {
        return(x-1)
    } else {
        return(x+1)
    }
}

nearestAdjacentY(y)
{
    /* Return Y coordinate of nearest adjacent location to object. */
    if (avatar.y == y) {
        return(y)
```

```
    } else if (avatar.y < y) {
        return(y-1)
    } else {
        return(y+1)
    }
}

noEffect()
{
    /* Action when nothing happens. */
    return
}

nullDestroy()
{
    /* Do nothing destruction. */
    return
}

nullInit()
{
    /* Do nothing initialization. */
    return
}

openClose()
{
    /* Open or close a door or gate, using key if neccessary. */
    haveKey = (holdingClass(KEY_CLASS) && avatar.inHand.keyNumber == self.key)
    if (!self.open && (haveKey || self.unlocked)) {
        @ self!OPEN () → (success)
        if (success) {
            self.open = TRUE
            self.unlocked = TRUE
            avatar.action = OPEN
        }
    } else if (self.open) {
        @ self!CLOSE () → (success)
        if (success) {
            self.open = FALSE
            if (haveKey) {
                self.unlocked = FALSE
            }
            avatar.action = CLOSE
        }
    } else {
        soundEffect(SOUND_THUMP)
    }
}

openCloseContainer()
{
    /* Open or close a container, using key if neccessary. */
    haveKey = (holdingClass(KEY_CLASS) && avatar.inHand.keyNumber == self.key)
```

```
    if (!self.open && (haveKey || self.unlocked)) {
        @ self!OPENCONTAINER () → (success, contentsVector)
        if (success) {
            unpackContentsVector(self, contentsVector)
            self.open = TRUE
            self.unlocked = TRUE
            avatar.action = OPEN_CONTAINER
        }
    } else if (self.open) {
        @ self!CLOSECONTAINER () → (success)
        if (success) {
            purgeContents(self)
            self.open = FALSE
            if (haveKey) {
                self.unlocked = FALSE
            }
            avatar.action = CLOSE_CONTAINER
        }
    }
}

oppositeX(x)
{
    /* Return the X coordinate on the opposite side of X from the
    avatar. */
    if (avatar.x == x) {
        return(x)
    } else if (avatar.x < x) {
        return(x+1)
    } else {
        return(x-1)
    }
}

oppositeY(y)
{
    /* Return the Y coordinate on the opposite side of Y from the
    avatar. */
    if (avatar.y == y) {
        return(y)
    } else if (avatar.y < y) {
        return(y+1)
    } else {
        return(y-1)
    }
}

payCoinOpOrDrop(cost, soundCoinDrop, soundSucceed, soundFail, goFunction)
{
    /* Handle coin operated devices.  Take the money, if you can, make the
    appropriate sounds, and return a boolean that tells whether or not we
    were successful in getting the device to operate. */
    if (goFunction()) {
        if (holdingClass(TOKEN_CLASS)) {
```

```
            if (avatar.inHand.denomination >= cost) {
                soundEffect(soundCoinDrop)
                @ self!PAY () → (success, text)
                avatar.action = PAY
                if (success) {
                    spend(cost)
                    soundEffect(soundSucceed)
                    return(TRUE)
                } else {
                    soundEffect(soundFail)
                    return(FALSE)
                }
            }
        } else {
            dropAt()
        }
    }
    return(FALSE)
}

payMe(amount)
{
    if (emptyHanded(avatar)) {
        createObject(TOKEN_CLASS, avatar, NULL, HAND)
        avatar.inHand.denomination = amount
    } else if (holdingClass(TOKEN_CLASS) {
        avatar.inHand.denomination += amount
    } else {
        newObject = createObject(TOKEN_CLASS, region, avatar.x, avatar.y)
        newObject.denomination = amount
    }
}

playMusic(score)
{
    /* *** Start up the sound driver playing the sequence of musical notes
    denoted by score *** */
}

playSpeech(text)
{
    /* *** Start up the sound driver playing the "speech" denoted by
    text. *** */
}

putInto(container, posX, posY, action)
{
    /* Put the object in the avatar's hand into the specified container at
    the specified location in the specified manner. */
    if (emptyHanded(avatar) && load(container) < lookupClass(container.class).
            capacity) {
        @ avatar.inHand!PUT (container.noid, posX, posY) → (success)
        if (success) {
            avatar.action = action
```

```
            subjectObject = avatar.inHand
            changeContainers(avatar.inHand, container, posX, posY)
        }
    }
}

pickObject(contentsList)
{
    /* *** Pop into rummage mode, rummaging among the items in
    contentsList, let the player pick one object (or delve deeper as
    suits him), and return a pointer to this object. *** */
}

purgeContents(object)
{
    /* Remove all knowledge of the contents of object. */
    for (i=0; i<(lookupClass(object.class).capacity; ++i) {
        purgeContents(object.contents[i])
        destroyObject(object.contents[i])
        object.contents[i] = NULL
    }
}

refund(amount, client)
{
    /* Give some money back in hand. */
    if (emptyHanded(client)) {
        createObject(TOKEN_CLASS, client, NULL, HAND)
        client.inHand.denomination = amount
    } else if (client.inHand.class == TOKEN_CLASS) {
        client.inHand.denomination += amount
    } else {
        newToken = createObject(TOKEN_CLASS, region, client.x, client.y)
        newToken.denomination = amount
    }
}

remove()
{
    /* If wearing, take off, else go and drop. */
    if (wearing(self)) {
        if (emptyHanded(avatar)) {
            changeContainer(self, avatar, NULL, HAND)
            avatar.action = undress
        }
    } else if (holding(self)) {
        dropHere()
    } else {
        goToAndDropAt()
    }
}

sendMail(checkAddress)
{
```

```
    /* If avatar is carrying a letter (i.e., a piece of paper with
    (presumably) an address and a message on it) it is taken from the
    avatar's hand and mailed away.  If carrying anything else, it is
    dropped in front of the mailbox. */
    if (holdingClass(PAPER_CLASS) && (checkAddress==NO_ADDRESS_CHECK ||
                self.address==avatar.address) {
        if (goTo()) {
            @ self!SENDMAIL (avatar.inHand.noid, avatar.inHand.text) → ()
            destroyObject(avatar.inHand)
        }
    } else {
        goToAndDropAt()
    }
}

setDownOrGetUp(posture)
{
    /* If sitting in seat, stand up.  If standing up, sit down if you can.
    Handles multiple place seating (couch, hot tub, etc.). */
    /* *** should reflect slot pointed to; add host interaction *** */
    for (i=0; i<selfClass.maxOccupants; ++i) {
        if (self.occupants[i] == NULL) {
            theSlot = i
        } else if (self.occupants[i] == avatar) {
            theSlot = i
            break
        }
    }
    if (adjacent(self) && i!=selfClass.maxOccupants) {
        if (goOnto()) {
            avatar.activity = posture
            self.occupants[theSlot] = avatar
        }
    } else if (here(self)) {
        if (goPreferred()) {
            avatar.activity = STANDING
            self.occupants[theSlot] = NULL
        }
    } else {
        depends()
    }
}

shoot()
{
    /* Attack somebody or something with a firearm. */
    if (!self.safetyOn) {
        soundEffect(SOUND_BANG)
        avatar.action = SHOOT
        if (target.class == AVATAR_CLASS) {
            @ self!ATTACK (target.noid) → (damage)
            soundEffect(SOUND_ATTACK)
            damageAvatar(target, damage)
        } else {
```

```
            @ self!BASH (target.noid) → (success)
            soundEffect(SOUND_BASH)
            if (success) {
                destroyObjectDramatically(target)
            }
        }
    } else {
        soundEffect(SOUND_DUD)
    }
}

sitOrGetUp()
{
    /* If sitting in seat, stand up.  If standing up, sit down if you can. */
    setDownOrGetUp(SITTING)
}

smallExplosion()
{
    /* When items of minor magic or volatile content are destroyed. */
    explosionAt(self.x, self.y, SMALL_EXPLOSION)
}

smallExplosionIfMagic()
{
    /* Certain items are only sometimes magical, and only the magical ones
   explode when you destroy them. */
    if (self.magic) {
        smallExplosion()
    } else {
        smash()
    }
}

smash()
{
    /* When ordinary items are destroyed. */
    explosionAt(self.x, self.y, SMASH_UP)
}

smashWithContents()
{
    /* When a container is destroyed, we have to destroy the stuff in it
   to, so if it is carrying a explodable that should explode. */
    for (i=self.contents; i!=NULL; i=i->next) {
        (i.destroy)()
    }
    smash()
}

soundEffect(sound)
{
    /* Emit a particular sound. */
    /* In Randy's sound driver code. */
```

```
}

spend(amount)
{
    /* Spend some number of tokens out of hand. */
    avatar.inHand.denomination -= amount
    if (avatar.inHand.denomination == 0) {
        destroyObject(avatar.inHand)
    }
}

strike()
{
    /* Attack somebody or something with a close-range melee weapon. */
    if (!elsewhere(target)) {
        avatar.action = STRIKE
        subjectObject = self
        targetObject = target
        if (target.class == AVATAR_CLASS) {
            @ self!ATTACK (target.noid) → (damage)
            soundEffect(SOUND_ATTACK)
            damageAvatar(target, damage)
        } else {
            @ self!BASH (target.noid) → (success)
            soundEffect(SOUND_BASH)
            if (success) {
                destroyObjectDramatically(target)
            }
        }
    }
}

textEdit(paper)
{
    /* *** Pop into text editor mode to view and edit the text on
   paper. *** */
}

textEditReadOnly(text)
{
    /* *** Pop into text editor mode to view text, but don't let the
   player alter what he sees. *** */
}

throw()
{
    /* Throw hand-held item at object's location.  Throw is called from
    under a reversed-do, so the we are now the object being thrown,
    and target tells us the destination.  It is rather like drop,
    except we don't know for sure where the object will land until the host
    tells us, and hitting something can do damage. */
    @ self!THROW (target.noid) → (newTarget, newX, newY, hit)
    avatar.action = THROW
    subjectObject = self
```

```
    changeContainers(self, newTarget, newX, newY)
    if (hit) {
        soundEffect(SOUND_HIT)
        /* *** effect of hit? *** */
    }
}

toggleSwitch(okToProceed)
{
    /* Turn the object on or off. */
    if (okToProceed) {
        if (self.on) {
            self.on = FALSE
            @ self!OFF () → ()
            soundEffect(SOUND_OFF)
        } else {
            self.on = TRUE
            @ self!ON () → ()
            soundEffect(SOUND_ON)
        }
        return(TRUE)
    } else {
        return(FALSE)
    }
}

unpackContentsVector(object, contentsVector)
{
    /* Decode contentsVector as the new contents of the given
   object. */
    /* In Randy's communications code. */
}

wear()
{
    /* If holding, put it on.  If not, go get it. */
    if (!wearing(self)) {
        if (!holding(self)) {
            goToAndGet()
        } else {
            changeContainer(self, avatar, NULL, selfClass.location)
            avatar.action = WEAR
        }
    }
}

wearing(clothing)
{
    /* Return true if avatar is wearing the item of clothing. */
    return(amongContents(clothing, avatar))
}

wearOrUnpocket()
{
```

```
    /* If wearing, remove item from pockets, else wear. */
    if (!wearing(self)) {
        wear()
    } else {
        get(pickObject(self.contents))
    }
}

writeTo(paper, text)
{
    /* Add the given text to the text on paper. */
    paper.text = concatenateStrings(paper.text, text)
}


ANSWER* (avatar, success)
{
    avatar.action = UNHOOK
    if (success) {
        self.state = PHONE_TALKING
        soundEffect(SOUND_SILENCE)
    } else {
        soundEffect(SOUND_LINE_THUMP)
        if (self.class == PHONE_BOOTH_CLASS) {
            self.state = PHONE_OFF_HOOK
        } else {
            self.state = PHONE_ACTIVE
        }
        soundEffect(SOUND_DIAL_TONE)
    }
}

ANSWERED* ()
{
    self.state = PHONE_TALKING
    soundEffect(SOUND_SILENCE)
}

DIAL* (success)
{
    if (success) {
        self.state = PHONE_LINE_RING
        soundEffect(SOUND_LINE_RING)
    } else {
        self.state = PHONE_LINE_BUSY
        soundEffect(SOUND_LINE_BUSY)
    }
}

DRIVE* (x, y)
{
    /* Asynchronous function when directed to move to another location and
   traveling in a powered vehicle. */
    self.targetX = x
```

```
    self.targetY = y
    self.moving = TRUE
    soundEffect(SOUND_ENGINE)
}

HANG* ()
{
    if (self.class == PHONE_BOOTH_CLASS && (self.state == PHONE_LINE_BUSY ||
            self.state == PHONE_LINE_RING)) {
        soundEffect(SOUND_JANGLE)
        refund(PHONE_CALL_COST, self.container)
    }
    soundEffect(SOUND_THUMP)
    soundEffect(SOUND_SILENCE)
    self.state = PHONE_READY
    self.container.action = HANGUP
}

HUNGUP* ()
{
    /* Asynchronous message from host when the party at the other end hangs
    up.  If we have answered we should get disconnected.  If we have not,
    then the phone should just stop ringing */
    if (self.state == PHONE_TALKING) {
        soundEffect(SOUND_LINE_THUMP)
        if (self.class == PHONE_BOOTH_CLASS) {
            self.state = PHONE_OFF_HOOK
        } else {
            self.state = PHONE_ACTIVE
        }
        soundEffect(SOUND_DIAL_TONE)
    } else {
        self.state = PHONE_READY
        soundEffect(SOUND_SILENCE)
    }
}

OFF* ()
{
    self.on = FALSE
}

OFFLIGHT* ()
{
    self.on = FALSE
    lightLevel--
}

ON* ()
{
    self.on = TRUE
}

ONLIGHT* ()
```

```
{
    self.on = TRUE
    lightLevel++
}

ORACLESPEAK+ (text)
{
    /* Asynchronous message from host when an oracle says something */
    /* *** Trigger this. *** */
    balloonMessage(self, "%s", text)
}

RING* ()
{
    /* Asynchronous message from host when the phone should ring.  Host
    makes sure we don't get this happening when the phone is not ready, so
    it *should* be safe to assume that the phone is in the PHONE_READY
    state */
    self.state = PHONE_RINGING
    soundEffect(SOUND_BELL_RING)
}

SPEAK* (text)
{
    /* Asynchronous message from host when player somebody says
    something. */
    balloonMessage(self, "%s", text)
}

UNHOOK* (avatar)
{
    if (self.class == PHONE_BOOTH_CLASS) {
        self.state = PHONE_OFF_HOOK
    } else {
        self.state = PHONE_READY
    }
    avatar.action = UNHOOK
    soundEffect(SOUND_CLICK_CLUNK)
    soundEffect(SOUND_DIAL_TONE)
}

WALK* (x, y)
{
    /* Asynchronous function when directed to move to another location. */
    self.targetX = x
    self.targetY = y
    self.moving = TRUE
    self.activity = WALKING
}
```

Host System Standard Actions and Functions

The home system object behavior code includes messages to the host which direct certain actions. The below functions are shared among several different classes of objects and so are collected here, rather than with the descriptions of the objects themselves.

```
ANSWER () → (success)
{
    if (!elsewhere(self) && self.state == PHONE_RINGING) {
        if (self.caller != NULL) {
            self.state = PHONE_TALKING
            self.caller.state = PHONE_TALKING
            talker = avatar
            success = TRUE
            # self → ANSWER* (avatar.noid, TRUE)
            self.caller → ANSWERED* ()
        } else {
            success = FALSE
        }
    } else {
        success = FALSE
    }
}

ASKORACLE (text) → ()
{
    if (self.live) {
        askQuestionLive(text)
    } else {
        newQuestion = alloc(QUESTION_ENTRY)
        newQuestion->question = text
        newQuestion->previousQuestion = self.questions
        self.questions = newQuestion
    }
}

ATTACK (targetId) → (damage)
{
    target = ↑targetId
    /* *** handle path check on ranged weapons *** */
    if ((!elsewhere(target) || isRangedWeapon(self)) && target.class ==
            AVATAR_CLASS) {
        damage = damageAvatar(target, self)
        # avatar → ATTACK* (targetId, damage)
    } else {
        damage = 0
    }
}

BASH (targetId) → (success)
{
    target = ↑targetId
    /* *** handle path check on ranged weapons *** */
    if ((!elsewhere(target) || isRangedWeapon(self)) && target.class !=
            AVATAR_CLASS) {
        success = damageObject(target, self)
        # avatar → BASH* (targetId, success)
    } else {
        success = FALSE
    }
```

```
}

CLOSE () → (success)
{
    haveKey = (holdingClass(KEY_CLASS) && avatar.inHand.keyNumber == self.key)
    if (!elsewhere(self) && self.open) {
        success = TRUE
        self.open = FALSE
        self.unlocked = !haveKey
        # avatar → CLOSE* (self, self.unlocked)
    } else {
        success = FALSE
    }
}

CLOSECONTAINER () → (success)
{
    haveKey = (holdingClass(KEY_CLASS) && avatar.inHand.keyNumber == self.key)
    if (!elsewhere(self) && self.open) {
        disappearContents(self)
        self.open = FALSE
        self.unlocked = !haveKey
        # avatar → CLOSECONTAINER* (self, self.unlocked)
        success = TRUE
    } else {
        success = FALSE
    }
}

DIAL (text) → (success)
{
    success = FALSE
    if (!elsewhere(self) && self.state == PHONE_ACTIVE) {
        self.talker = avatar
        self.caller = lookupPhoneNumber(text)
        if (self.caller != NULL) {
            if (self.caller.state == PHONE_READY) {
                self.caller.state = PHONE_RINGING
                self.caller.caller = self
                self.state = PHONE_LINE_RING
                self.caller → RING* ()
                if (self.caller.owner.container != self.caller.container) {
                    if (self.caller.beeper != NULL) {
                        self.caller.beeper → BEEP* ()
                    }
                    if (self.caller.answeringMachine != NULL) {
                        answerPhoneWithMachine(self.caller.answeringMachine)
                    }
                }
                success = TRUE
            } else {
                self.state = PHONE_LINE_BUSY
            }
        } else {
```

```
            success = FALSE
        }
        # self → DIAL* (success)
    } else {
        success = FALSE
    }
}

GET () → (success)
{
    if (emptyHanded(avatar) && accessible(self)) {
        changeContainers(self, avatar, NULL, HAND)
        if (self.class == BEEPER_CLASS) {
            avatar.homePhone.beepr = self
        } else if (self == avatar.homePhone.answeringMachine) {
            avatar.homePhone.answeringMachine = NULL
        }
        success = TRUE
        # avatar → GET* (self.noid)
    } else {
        success = FALSE
    }
}

HANG () → ()
{
    if (!elsewhere(self) && self.state != PHONE_READY) {
        self.state = PHONE_READY
        talker = NULL
        if (self.caller != NULL) {
            self.caller.caller = NULL
            self.caller → HUNGUP* ()
        }
        # self → HANG* ()
        self.caller = NULL
    }
}

MAGIC () → ()
{
    (*magicBehavior[self.magicType])() /* *** expand *** */
}

OFF () → ()
{
    if (!elsewhere(self) && self.on) {
        self.on = FALSE
        # self → OFF* ()
    }
}

OFFPLAYER () → ()
{
    if (!elsewhere(self) && self.on) {
```

```
        self.on = FALSE
        # self → OFF* ()
        dequeuePlayer(self)
    }
}

ON () → ()
{
    if (!elsewhere(self) && !self.on) {
        self.on = TRUE
        # self → ON* ()
    }
}

ONPLAYER () → ()
{
    if (!elsewhere(self) && !self.on) {
        self.on = TRUE
        # self → ON* ()
        enqueuePlayer(self)
    }
}

OPEN () → (success)
{
    haveKey = (holdingClass(KEY_CLASS && avatar.inHand.keyNumber == self.key)
    if (!elsewhere(self) && !self.open && (haveKey || self.unlocked)) {
        self.open = TRUE
        self.unlocked = TRUE
        success = TRUE
        # avatar → OPEN* (self)
    } else {
        success = FALSE
    }

}

OPENCONTAINER () → (success, contentsVector)
{
    haveKey = (holdingClass(KEY_CLASS) && avatar.inHand.keyNumber == self.key)
    if (!elsewhere(self) && !self.open && (haveKey || self.unlocked)) {
        contentsVector = vectorize(self.contents)
        self.open = TRUE
        self.unlocked = TRUE
        success = TRUE
        # avatar → OPENCONTAINER* (self, contentsVector)
    } else {
        success = FALSE
        contentsVector = ""
    }
}

PUT (containerId, posX, posY) → (success)
{
```

```
    container = ↑containerId
    if (holding(self) && available(container, posX, posY) && load(container) <
            lookupClass(container.class).capacity) {
        changeContainers(self, container, posX, posY)
        success = TRUE
        # avatar → PUT* (self.noid, containerId, posX, posY)
        if (self.class == BEEPER_CLASS) {
            avatar.homePhone.beeper = NULL
        } else if (self.class == ANSWERING_MACHINE_CLASS) {
            if (self.container == avatar.homePhone.container) {
                avatar.homePhone.answeringMachine = self
            }
        }
    } else {
        success = FALSE
    }
}

SENDMAIL (paperId, text) → ()
{
    paper = ↑paperId
    if (paper.class == PAPER_CLASS) {
        sendMailMessage(text)
        destroyObject(paper)
    }
}

TALK (text) → ()
{
    if (!elsewhere(self) && self.state == PHONE_ACTIVE && self.caller != NULL) {
        self.caller → SPEAK* (text)
    }
}

THROW (targetId) → (newTarget, newX, newY, hit)
{
    target = ↑targetId
    if (holding(self)) {
        checkPath(target, target.x, target.y, &newX, &newY)
        hit = (newX == target.x && newY == target.y)
        if (hit) {
            newTarget = targetId
            # avatar → THROW* (targetId, newX, newY, hit)
        } else {
            newTarget = region.noid
            # avatar → THROW* (region.noid, newX, newY, hit)
        }
    } else {
        newX = self.x
        newY = self.y
        hit = FALSE
    }
}
```

```
UNHOOK () → ()
{
    if (!elsewhere(self) && self.state == PHONE_READY) {
        if (self.class == PHONE_BOOTH_CLASS) {
            self.state = PHONE_OFF_HOOK
        } else {
            self.state = PHONE_ACTIVE
        }
        # self → UNHOOK* (avatar.noid)
    }
}

accessible(object)
{
    /* *** Return TRUE if avatar can get object. Needs, among
    other things, to handle interlock used for countertop and display
    case. *** */
}

alloc(type)
{
    /* Allocate storage for an entity of the given type, and return a
    pointer to this storage. */
    /* Built into PL/1. */
}

announceObject(object)
{
    /* *** Announce the existence of object and its state to
    everybody in the region except the ourselves. *** */
}

answerPhoneWithMachine(phone, machine)
{
    /* *** Somebody is trying to call phone, but nobody's home, so
    have machine answer, play its message, and take a message from
    the caller. *** */
}

askQuestionLive(text)
{
    /* Send text to the system person currently playing
    oracle. *** Ask JH *** */
}

atWater()
{
    /* *** Return TRUE if there is water at the avatar's present
    location. *** */
}

available(container, x, y)
{
    /* Return TRUE if there is not already an object occupying
```

```
    the given (x,y) location inside container. */
     return(container.contents[y] == NULL)
}

cancelEvent(eventId)
{
     /* Cancel the previously scheduled event indicated by
    eventId. *** Ask JH *** */
}

changeContainers(object, newContainer, newX, newY)
{
     /* Move object from wherever it is now to the indicated
    (x, y) location in the indicated new container. */
     object.container.contents[object.y] = NULL
     object.container = newContainer
     object.x = newX
     object.y = newY
     newContainer.contents[newY] = object
}

checkPath(target, x, y, newx, newy)
{
     /* Check for a path from the avatar's present location to the
    object target at the given (x,y) location in the region. Set
    newx and newy to the (x,y) location of the farthest point
    from the avatar along that path which can be reached in a straight line
    (of course, if the way is completely clear, newx=x and
    newy=y. *** Discuss with JH *** */
}

createObject(class, container, x, y)
{
     /* Create and object of the specified class located at the given
    (x,y) location within the given container. Return a pointer to the new
    instance descriptor. *** Ask JH *** */
}

damageAvatar(who, damage)
{
     /* *** When an avatar is injured, subtract damage from his health
    and apply the appropriate effects accordingly. *** */
}

damageObject(object, weapon)
{
     /* *** Return TRUE iff attacking object with weapon
    damages it. *** */
}

dequeuePlayer(player)
{
     /* Remove the given player object (radio or tape deck) from the
    queue of things receiving PLAY+() messages. *** Ask JH *** */
```

```
}

destroyContents(object)
{
    /* Destroy all the objects that object contains. */
    for (i=0; i<(lookupClass(object.class).capacity; ++i) {
        destroyObject(object.contents[i])
        object.contents[i] = NULL
    }
}

destroyObject(object)
{
    /* Destroy the given object.  I.e., delete it and garbage collect
    all its resources. *** Ask JH *** */
}

disappearContents(container)
{
    /* *** As far as the players are concerned, delete all the objects
    contained in container.  As far as we are concerned, these
    objects continue to exist. *** */
}

disappearObject(object)
{
    /* *** As far as the players are concerned, delete object.  As
    far as we are concerned, it continues to exist. *** */
}

elsewhere(object)
{
    /* Test if object is elsewhere.  To be elsewhere, object must be
    neither where the avatar is nor adjacent to him nor on his person. */
    return(!here(object) && !adjacent(object) && object.container!=avatar)
}

emptyHanded(who)
{
    /* Return TRUE iff the avatar is not holding anything. */
    return(who.inHand == NULL)
}

enqueuePlayer(player)
{
    /* Add the given player (radio or tape deck) to the queue of things
    that need to be send PLAY+() messages every so often. *** Ask JH *** */
}

holding(object)
{
    /* Return TRUE iff the avatar is holding the given object. */
    return(avatar.inHand == object)
}
```

```
holdingClass(class)
{
     /* Return TRUE iff the avatar is holding an object of class class */
     return(!emptyHanded(avatar) && avatar.inHand.class == class)
}

goToNewRegion(avatar, region)
{
     /* Transport avatar to region region. *** Ask JH *** */
}

isRangedWeapon(object)
{
     /* *** Return TRUE iff object is a ranged weapon (e.g., a
    gun). *** */
}

load(object)
{
     /* Compute the load inside object, i.e., how many things it
    is holding. */
     result = 0
     for (i=0; i<(lookupClass(object.class).capacity; ++i) {
         if (object.contents[i] != NULL) {
             result++
         }
     }
     return(result);
}

lookupClass(classNumber)
{
     /* Return the class descriptor associated with the given class
    number. */
     /* *** Ask JH *** */
}

lookupPhoneNumber(text)
{
     /* *** Try to interpret text as a phone number and return a
    pointer to the telephone object that the number corresponds to.
    (NULL if there is none.) *** */
}

lookupSelection(musicList, listLength, text)
{
     /* *** Text is a jukebox selection.  Figure out which piece in
    musicList it is and return a pointer to it. *** */
}

lookupTeleportAddress(text)
{
     /* *** Try to interpret text as a teleport number and return a
    pointer to the teleport object that the number corresponds to.
```

```
    (NULL if there is none.) *** */
}

payMe(amount)
{
    /* Pay the given amount to the avatar. */
    payTo(avatar, amount)
}

payTo(player, amount)
{
    /* Pay the given amount to the designated player. */
    if (emptyHanded(player)) {
        createObject(TOKEN_CLASS, player, NULL, HAND)
        player.inHand.denomination = actualWithdrawal
        announceObject(player.inHand)
    } else if (player.inHand.class == TOKEN_CLASS) {
        player.inHand.denomination += actualWithdrawal
    } else {
        newObject = createObject(TOKEN_CLASS, region, player.x, player.y)
        newObject.denomination = actualWithdrawal
        announceObject(newObject)
        }
    }
}

random()
{
    /* Return a random integer. *** Ask JH *** */
}

randomTimeInTheFuture()
{
    /* *** Return some random time in the future. *** */
}

scheduleEvent(eventProcedure, delay)
{
    /* Set things up so that the procedure eventProcedure will be
    called delay time units from now.  Return an identifier of some
    sort so we can cancel this in the future if we so desire.
    *** Ask JH *** */
}

sendMailMessage(text)
{
    /* Interpret text as a mail message and send it off.  We
    extract the address from the text itself. *** Ask JH *** */
}

spend(cost)
{
    /* Have the avatar spend the given amount out of pocket or bank
    account, if he can.  Return a boolean indicating how successful this
```

```
    was. */
     if (holdingClass(TOKEN_CLASS) && avatar.inHand.denomination >= cost) {
         avatar.inHand.denomination -= cost
         if (avatar.inHand.denomination == 0) {
             destroyObject(avatar.inHand)
         }
         return(TRUE)
     } else if (holdingClass(CREDIT_CARD_CLASS) && avatar.bankAccountBalance >=
             cost) {
         avatar.bankAccountBalance -= cost
         return(TRUE)
     } else {
         return(FALSE)
     }
}

vectorize(contentsList)
{
     /* Take the given contents list and turn it into a byte vector
    suitable for transmission to the players. *** Ask JH *** */
}
```

The Objects

The object descriptions follow on the proceeding pages:

Object:

amulet

Description:

A hang-around-the neck type amulet.

Function:

Generic magic talisman.

Notes:

Amulets are always magical. An amulet will do something special and powerful. The magical function may vary. ***** The set of magical functions needs thought. ***** Amulets are particularly rare and contain unusually powerful forms of magic. The general function of an amulet will be to confer some particular power or ability on the avatar wearing it.

Styles:

None.

Properties:

No properties.

Class properties:

```
location = NECK /* Where on the body this is worn. */
```

Host properties:

```
magicType       /* What sort of magic this amulet contains.  Used as case
                   switch for execution of magical behavior. */
```

Command Behavior:

Do:

```
doMagic()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
wear()
```

Put:

```
remove()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
nullInit()
```

Destruction:

```
smallExplosion()
```

Asynchronous actions:

None.

Graphics:

Single static image of amulet sitting by itself. Cels for amulet around avatar's neck in front and side views.

Host behavior:

```
GET
MAGIC
PUT
THROW
```

Object:

answering machine

Description:

A telephone answering machine.

Function:

Handle your phone calls when you're not at home.

Notes:

The answering machine operates in conjunction with a telephone (of the non-pay and non-portable variety, of which there may be only one per region). Most of the answering machine's guts reside in the host. The answering machine has a light on it that is lit if there are messages waiting.

Styles:

None.

Properties:

```
waitingMessages /* A flag telling if there are waiting messages.  TRUE if
                   there are. */
```

Class properties:

No class properties.

Host properties:

```
messageQueue    /* The waiting messages themselves, if there are any, in FIFO
                   order. */
owner           /* The avatar who belongs to this machine. */
```

Command Behavior:

Do:

```
{
    /* If there is a message waiting, it is played back.  Note that the
    host will allow only the machine's owner to play the messages. */
    if (!elsewhere(self)) {
        if (self.messageWaiting) {
            @ self!PLAYMESSAGE () → (moreMessages, text)
            self.messageWaiting = moreMessages
            balloonMessage(self, "%s", text)
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
{
    /* Use the typed text string as the new message on the machine. */
    @ self!SETANSWER (text) → ()
}
```

Reversed Do:

```
noEffect()
```

Initialization:

```
init(waitingMessages)
{
```

```
        self.waitingMessages = waitingMessages
    }
```

Destruction:

```
    smash()
```

Asynchronous actions:

```
    TAKEMESSAGE* ()
    {
        /* Asynchronous action when a message is taken. */
        /* *** Trigger this. *** */
        self.messagesWaiting = TRUE
    }
```

Graphics:

Two state image of the machine: with message light off and with message light on.

Host behavior:

```
    GET
    PLAYMESSAGE () → (moreMessages, text)
    {
        if (self.owner == avatar) {
            if (self.messageQueue->firstMessage != NULL) {
                text = self.messageQueue->firstMessage->text
                deadMessage = self.messageQueue->firstMessage
                self.messageQueue = self.messageQueue->firstMessage->nextMessage
                garbageCollect(deadMessage)
            } else {
                text = ""
            }
            moreMessages = (self.messageQueue->firstMessage != NULL)
            if (!moreMessages) {
                self.messageQueue->lastMessage = NULL
            }
        }
    }
    PUT
    SETANSWER (text) → ()
    {
        if (self.owner == avatar) {
            newMessage = alloc(MESSAGE_QUEUE_ENTRY)
            newMessage->text = text
            newMessage->nextMessage = NULL
            if (self.messageQueue->lastMessage == NULL) {
                self.messageQueue->firstMessage = newMessage
            } else {
                self.messageQueue->lastMessage->nextMessage = newMessage
            }
            self.messageQueue->lastMessage = newMessage
        }
    }
```

Object:
aquarium
Description:
Your basic fish tank.
Function:
Household decoration.
Notes:
The aquarium is just a visual joke.
Styles:
None.
Properties:

```
state           /* The state that the aquarium is currently in.  The possible
                   states are FISH_SWIMMING, FISH_FEEDING, FISH_DEAD and
                   FISH_GONE. */
```

Class properties:
No class properties.
Host properties:

```
fishDie         /* Event that will kill off the fish if you don't feed them in
                   time. */
```

Do:

```
{
    /* Feed the fish, if you have the fish food. */
    if (holdingClass(FISH_FOOD_CLASS) && adjacent(self) && self.state ==
            FISH_SWIMMING) {
        @ self!FEED () → ()
        self.state = FISH_FEEDING
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
GoToAndGet()
```

Put:

```
GoToAndDropAt()
```

Talk:

```
broadcast
```

Reversed Do:

```
throw()
```

Initialization:

```
init(state)
{
    self.state = state
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
CHANGESTATE* (newState)
{
    /* Asynchronous message from host when state of fish changes. */
    self.state = newState
```

```
}
```

Graphics:

Animation of fish swimming in tank. Animation of fish feeding. Static image of dead fish floating at top of tank. Static image of empty tank.

Host behavior:

```
FEED () → ()
{
    if (holdingClass(FISH_FOOD_CLASS) && adjacent(self) && self.state ==
            FISH_SWIMMING) {
        self.state = FISH_FEEDING
        # self → CHANGESTATE* (self.state)
        cancelEvent(self.fishDie)
        self.fishDie = NULL
        scheduleEvent(FishFed, FISH_FEED_DELAY)
    }
}
FishFed()
{
    self.state = FISH_SWIMMING
    # self → CHANGESTATE* (self.state)
    self → CHANGESTATE* (self.state)
    self.fishDie = scheduleEvent(FishDie, FISH_DEATH_DELAY)
}
FishDie()
{
    self.state = FISH_DEAD
    self.fishDie = NULL
    # self → CHANGESTATE* (self.state)
    self → CHANGESTATE* (self.state)
}
```

Object:

atm

Description:

Automatic **T**oken **M**achine.

Function:

Dispenses tokens.

Notes:

Every player in the **MicroCosm** has a bank account at the First MicroCosm Bank & Trust Company. Money that is in your bank account cannot be taken from you by force (i.e., it doesn't protect you from blackmail, fraud, etc.). The ATM is a device for accessing the account.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
{
    /* Causes the machine to tell you your account balance */
    balloonMessage(self, "Balance: T%d", bankAccountBalance)
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
{
    /* Takes 50 tokens or your entire balance, whichever is less, from
    your account and puts them in your avatar's hand. */
    if (goTo()) {
        if (bankAccountBalance < 50) {
            withdrawal = bankAccountBalance
        } else {
            withdrawal = 50
        }
        if (withdrawal > 0) {
            @ self!WITHDRAW (withdrawal) → (actualWithdrawal)
            soundEffect(SOUND_WITHDRAW)
            bankAccountBalance -= actualWithdrawal
            balloonMessage(self, "New balance: T%d", bankAccountBalance)
            payMe(actualWithdrawal)
        } else {
            balloonMessage(self, "Insufficient funds")
        }
    }
}
```

Put:

```
{
    /* If item in hand is tokens, these are deposited in your account and
    disappear from hand, otherwise item is dropped in front of machine */
    if (goTo()) {
```

```
            if (holdingClass(TOKEN_CLASS)) {
                @ self!DEPOSIT (avatar.inHand.noid) → ()
                soundEffect(SOUND_DEPOSIT)
                bankAccountBalance += avatar.inHand.denomination
                balloonMessage(self, "New balance: T%d", bankAccountBalance)
                destroyObject(avatar.inHand)
            } else {
                dropAt();
            }
        }
    }
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
nullInit()
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static cel image of machine. Sound effect of money being withdrawn. Sound effect of money being deposited.

Host behavior:

```
DEPOSIT (tokenId) → ()
{
    token == ↑tokenId
    if (holding(token) && token.class == TOKEN_CLASS) {
        bankAccountBalance += token.denomination
        destroyObject(token)
    }
}
WITHDRAW (withdrawal) → (actualWithdrawal)
{
    if (bankAccountBalance >= withdrawal) {
        actualWithdrawal = withdrawal
    } else {
        actualWithdrawal = bankAccountBalance
    }
    bankAccountBalance -= actualWithdrawal
    if (actualWithdrawal > 0) {
        payTo(avatar, actualWithdrawal);
    }
}
```

Object:

avatar

Description:

You or somebody else.

Function:

The animated figure.

Notes:

The avatar object is one of the keystones of the system. Everything that happens is oriented around one or another avatars performing some action. As a result, the state information associated with an avatar is quite complex.

Styles:

None.

Properties:

```
contents        /* The thing the avatar is wearing or holding.  This is
                   usually not referenced directly, but through the following
                   specific locations (the X position ''inside'' the avatar is
                   used to identify them): */
inHand          /* The object that the avatar has in its hands.  NULL if hands
                   are empty.  This is part of contents, X == 0. */
torso           /* What the avatar is wearing on its torso (shirt, jacket,
                   etc.).  NULL if not wearing anything there.  This is part
                   of contents, X == 1. */
legs            /* What the avatar is wearing on its legs (pants, skirt,
                   etc.). NULL if not wearing anything there.  This is part of
                   contents, X == 2. */
feet            /* What the avatar is wearing on its feet (shoes, boots,
                   etc.).  NULL if not wearing anything there.  This is part
                   of contents, X == 3. */
head            /* What the avatar is wearing on its head (hat, helmet, etc.).
                   NULL if not wearing anything there.  This is part of
                   contents, X == 4. */
activity        /* On-going activity that the avatar is in the midst of
                   (WALKING, SITTING, STANDING, etc.) */
action          /* One-time action that the avatar is executing (JUMP, WAVE,
                   PICKUP, POUR, etc.) */
actionStep      /* What step of the above action the avatar is in. */
orientation     /* What direction the avatar is facing (WEST, EAST, NORTH or
                   SOUTH only). */
health          /* Health of the avatar in the range 0 to 255.  An avatar with
                   a health of 0 is dead. */
restrainer      /* If the avatar is restrained by another avatar, this will
                   tell who the restrainer is.  If not, this will be NULL. */
address         /* Mail address associated with this avatar. */
moving          /* A flag that the avatar is in motion. */
targetX         /* Motion destination X position. */
targetY         /* Motion destination Y position. */
```

Class properties:

No class properties.

Host properties:

```
accountBalance  /* Balance in the avatar's bank account (in Tokens). */
player          /* Identification of player user-name associated with this
                   avatar. */
turf            /* Region that is the avatar's home turf. */
busFarePaid     /* Flag that avatar has paid to ride the bus. */
```

```
    homePhone       /* Object that is avatar's home phone. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
{
    /* If me, change posture, otherwise, goto. */
    if (avatar == self) {
        if (self.activity == STANDING) {
            self.activity = SITTING
            @ self!POSTURE (SITTING) → ()
        } else if (self.activity == SITTING) {
            self.activity = LAYING
            @ self!POSTURE (LAYING) → ()
        } else {
            self.activity = STANDING
            @ self!POSTURE (STANDING) → ()
        }
    } else {
        goTo()
    }
}
```

Stop:

```
cease()
```

Get:

```
{
    /* If not me, go to the other avatar and take whatever is in his hands.
    If his hands are empty then grab him and restrain him.  If it is me,
    then no effect. */
    /* Try not to get too confused: in this context avatar is the
    avatar object associated with the player, while self is the
    avatar object being pointed to. */
    if (self != avatar) {
        if (goTo()) {
            if (emptyHanded(avatar)) {
                @ self!GRAB () → (success, grabbedAvatar)
                if (success) {
                    if (!grabbedAvatar) {
                        changeContainers(self.inHand, avatar, NULL, HAND)
                    } else {
                        self.restrainer = avatar
                }
            }
        }
    } else {
        noEffect()
    }
}
```

Put:

```
{
    /* If not me, go to the other avatar give him whatever is in my hands.
    If it is me, no effect. */
    if (self != avatar) {
        if (goTo()) {
```

```
                if (emptyHanded(self)) {
                    @ self!HAND () → (success)
                    if (success) {
                        changeContainers(avatar.inHand, self, NULL, HAND)
                    }
                }
            }
        } else {
            noEffect()
        }
    }
```

Talk:

```
    {
        /* If not me, text message is sent to other avatar as a
        person-to-person message.  If me, broadcast. */
        if (self != avatar) {
            @ avatar!SPEAK (self.noid, text) → ()
        } else {
            broadcast()
        }
    }
```

Reversed Do:

```
    noEffect()
```

Initialization:

```
    init(activity, orientation, health, address, moving, targetX, targetY,
            contentsVector)
    {
        self.activity = activity
        self.action = NULL
        self.actionStep = 0
        self.orientation = orientation
        self.health = health
        self.restrainer = NULL
        self.address = address
        self.moving = moving
        self.targetX = targetX
        self.targetY = targetY
        unpackContentsVector(self, contentsVector)
    }
```

Destruction:

```
    killAvatar()
```

Asynchronous actions:

```
    ATTACK* (target, damage)
    {
        soundEffect(SOUND_ATTACK)
        self.action = STRIKE
        subjectObject = self.inHand
        targetObject = target
        damageAvatar(target, damage)
    }
    BASH* (target, success)
    {
        soundEffect(SOUND_BASH)
        self.action = STRIKE
```

```
        subjectObject = self.inHand
        targetObject = target
        if (success) {
            destroyObjectDramatically(target)
        }
    }
    BUGOUT* ()
    {
        soundEffect(SOUND_ZAP)
        escaper = self.inHand
        escaper.charge--
        if (escaper.charge == 0) {
            destroyObject(escaper)
        }
    }
    CLOSE* (door, unlocked)
    {
        door.open = FALSE
        door.unlocked = unlocked
        self.action = CLOSE
    }
    CLOSECONTAINER* (container, unlocked)
    {
        purgeContents(container)
        container.open = FALSE
        container.unlocked = unlocked
        self.action = CLOSE_CONTAINER
    }
    FILL* ()
    {
        self.inHand.filled = TRUE
        self.action = FILL
    }
    GET* (object)
    {
        changeContainers(object, self, NULL, HAND)
        self.action = PICKUP /* *** What if picking from container? *** */
        subjectObject = object
    }
    GRAB* (avatar)
    {
       avatar.restrainer = self
    }
    GRABFROM* (avatar)
    {
        changeContainers(avatar.inHand, self, NULL, HAND)
    }
    OPEN* (door)
    {
        door.open = TRUE
        door.unlocked = TRUE
        self.action = OPEN
    }
    OPENCONTAINER* (container, contentsVector)
```

```
    {
        unpackContentsVector(container, contentsVector)
        container.open = TRUE
        container.unlocked = TRUE
        self.action = OPEN_CONTAINER
    }
    PAYTO* (amount)
    {
        if (self == avatar) {
            bankAccountBalance += amount
        }
        self.action = RECEIVE
    }
    POSTURE* (newPosture)
    {
        self.activity = newPosture
    }
    POUR* ()
    {
        self.inHand.filled = FALSE
        self.action = POUR
    }
    PUT* (container, x, y)
    {
        subjectObject = self.inHand
        changeContainers(self.inHand, container, x, y)
        self.action = PUT
    }
    SPEAK*
    TAKE* ()
    {
        self.inHand.count--
        self.action = TAKE
        if (self.inHand.count == 0) {
            destroyObject(self.inHand)
        }
    }
    THROW* (target, x, y, hit)
    {
        self.action = THROW
        subjectObject = self.inHand
        changeContainers(self.inHand, target, x, y)
        if (hit) {
            soundEffect(SOUND_HIT)
            /* *** effect of hit? *** */
        }
    }
    WALK*
```

Graphics:

A complex set of animation images expressing a wide range of motions and actions. See the document *Avatar Animation Specifications* for more detail.

Host behavior:

```
    GRAB () → (success, grabbedAvatar)
    {
```

```
        success = FALSE
        grabbedAvatar = FALSE
        if (emptyHanded(avatar)) {
            success = TRUE
            if (self.inHand != NULL) {
                changeContainers(self.inHand, avatar, NULL, HAND)
                # avatar → GRABFROM* (self)
            } else {
                grabbedAvatar = TRUE
                self.restrainer = avatar
                # avatar → GRAB* (self)
            }
        }
    }
    HAND () → (success)
    {
        if (!emptyHanded(avatar) && emptyHanded(self)) {
            success = TRUE
            changeContainers(avatar.inHand, self, NULL, HAND)
            # self → GRABFROM* (avatar)
        } else {
            success = FALSE
        }
    }
    POSTURE (newPosture) → ()
    {
        if (self == avatar) {
            if (newPosture == SITTING || newPosture == LAYING || newPosture ==
                    STANDING) {
                avatar.activity = newPosture
                # avatar → POSTURE* (newPosture)
            }
        }
    }
    SPEAK (audience, text) → ()
    {
        if (audience == NULL) {
            # avatar → SPEAK (text)
        } else {
            self → SPEAK (text)
        }
    }
    WALK (x, y) → (destinationX, destinationY)
    {
        if (self != avatar) {
            destinationX = self.x
            destinationY = self.y
        } else {
            checkPath(region, x, y, &destinationX, &destinationY)
            if (destinationX != self.x || destinationY != self.y) {
                self.x = destinationX
                self.y = destinationY
                # self → WALK* (destinationX, destinationY)
            }
```

```
        }
    }
```

Object:

backpack

Description:

Your basic backpack.

Function:

Can carry multiple items without use of hands to hold them all.

Notes:

A backpack object is an object-carrier that an avatar can put on and take off at will. It has a large capacity, but otherwise works like an item of clothing with pockets.

Styles:

None.

Properties:

```
contents                /* A list of the objects in the backpack. */
```

Class properties:

```
location = BACK         /* Where on the body this is worn. */
capacity = ?            /* How many things this can carry. */
displayContents = FALSE /* Whether we should show what's in it. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
wearOrUnpocket()
```

Put:

```
remove()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(contentsVector)
{
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

Single static frame of lone backpack, plus additional animation cels of pack on torso of avatar in side, front and back views.

Host behavior:

```
GET
PUT
THROW
```

Object:
bag
Description:
The sack.
Function:
Can carry multiple items using only one hand.
Notes:
A bag is simply an object carrier that can be carried in the hands.
Styles:
None.
Properties:

```
contents        /* A list of the objects in the bag. */
open            /* Flag that bag is open. */
key             /* Key to unlock it (always null, since it has none). */
unlocked        /* Flag that it is unlocked (always TRUE). */
```

Class properties:

```
capacity = ?    /* How many things this can hold. */
displayContents = FALSE /* Whether to show what's in it. */
```

Host properties:
No other properties.
Command Behavior:
Do:

```
adjacentOpenCloseContainer()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndPickFromOrGet()
```

Put:

```
goToAndDropIntoIfOpen()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(open, contentsVector)
{
    self.open = open
    self.key = NULL
    self.unlocked = TRUE
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:
None.
Graphics:
Two state image of bag: full and empty.
Host behavior:

```
CLOSECONTAINER
GET
OPENCONTAINER
PUT
```

THROW

Object:

ball

Description:

Your standard spherical throwing toy.

Function:

For playing games.

Notes:

The ball is relatively inert. You can pick it up and throw it. It you throw it at an avatar who is not already holding something else, that avatar will catch it.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw() /* *** Need something special here. *** */
```

Initialization:

```
nullInit()
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image of ball (no matter how you look at it, it looks the same).

Host behavior:

```
GET
PUT
THROW
```

Object:

bed

Description:

An ordinary bed.

Function:

Can be laid upon.

Notes:

The bed is a type of scenic object, usually found in building interiors. Avatars can lay down on it. This doesn't really change anything, it's just for appearance.

Styles:

```
BED_SINGLE, BED_KING_SIZE, BED_BRASS
```

Properties:

```
orientation             /* How the bed is oriented w.r.t p.o.v.  It can be any
                           of BED_LEFT_SIDE, BED_RIGHT_SIDE, BED_FOOT or
                           BED_HEAD. */
occupants[BED_SIZE]     /* The avatars laying there.  NULL if nobody is. */
```

Class properties:

```
maxOccupants = BED_SIZE = 2
```

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    setDownOrGetUp(LAYING)
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(bedStyle, orientation, occupants[BED_SIZE])
{
    self.grstyle = bedStyle
    self.orientation = orientation
    for (i=0; i<BED_SIZE; ++i) {
        self.occupants[i] = occupants[i]
    }
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Three static images for each type of bed: side view, head view and foot view.

Host behavior:

None.

Object:

beeper

Description:

One of those annoying little devices that disrupts movie theaters.

Function:

Alerts you that someone is trying to call you on the phone.

Notes:

If a phone call arrives at your home phone when you are not there (regardless of whether or not you have an answering machine to pick it up), the beeper starts beeping.

Styles:

None.

Properties:

```
beeping          /* Flag that we are in the beeping state.  TRUE if we are. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    /* Press the button to stop the beeping */
    if (holding(self)) {
        if (self.beeping) {
            self.beeping = FALSE
            soundEffect(SOUND_SILENCE)
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(beeping)
{
    self.beeping = beeping
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
BEEP* ()
{
    /* Asynchronous action when somebody else gets beeped. */
    self.beeping = TRUE
```

```
        soundEffect(SOUND_BEEPING)
    }
```

Graphics:

Single static image of beeper. Sound of beeper beeping.

Host behavior:

GET
PUT
THROW

Object:

boat

Description:

A nice little runabout.

Function:

Carries avatars and their possessions across water.

Notes:

The boat is a vehicle that can travel on water (but, of course, not on land) — rivers, ponds, seas, and so on. It operates just like a car otherwise. It is a simple rowboat-type boat with a small outboard motor. It does not go very fast nor does it carry very many people.

Styles:

None.

Properties:

```
contents        /* A list of the objects in the boat.  NULL if there are
                   none. */
inhabitants[BOAT_SIZE] /* The folks riding in it.  There are four slots which
                   are NULL if unoccupied.  The avatar in slot 0 gets to
                   drive. */
moving          /* A flag that the boat is in motion. */
targetX         /* Motion destination X position. */
targetY         /* Motion destination Y position. */
```

Class properties:

```
capacity = ?                /* How many things this can hold. */
displayContents = FALSE     /* Whether to show what's in it. */
maxOccupants = BOAT_SIZE = 4 /* How many people can fit in it. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
climbInOrOut()
```

Stop:

```
cease()
```

Get:

```
goToAndPickFrom()
```

Put:

```
goToAndDropInto()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(moving, targetX, targetY, inhabitants[BOAT_SIZE], contentsVector)
{
    self.moving = moving
    self.targetX = targetX
    self.targetY = targetY
    for (i=0; i<BOAT_SIZE; ++i) {
        self.inhabitants[i] = inhabitants[i]
    }
    unpackContentsVector(self, contentsVector)
}
```

Destruction:
`smashWithContents()`
Asynchronous actions:
`DRIVE*`
Graphics:
Three static images of boat: side view, front view, back view. Sound of boat engine driving.
Host behavior:
None.

Object:

book/newspaper

Description:

A readable document.

Function:

Displays text or artwork on paper.

Notes:

A book's contents are kept in the host. These are downloaded (in pieces) on demand when the player tries to read them. We use the full-screen text editor routines that are used for mail and pieces of paper, but we use them in a read-only mode that allows us to handle larger documents (since the contents may be discarded after they have scrolled off the screen).

Styles:

```
DOC_BOOK, DOC_MAGAZINE, DOC_NEWSPAPER, DOC_FLYER
```

Properties:

```
position        /* Current position in the document. */
```

Class properties:

No class properties.

Host properties:

```
content         /* The text that makes up the content of the document. */
maximumLength   /* The size of the text. */
```

Command Behavior:

Do:

```
{
    /* If holding and not already reading, start reading.  If already
    reading, read some more. */
     if (holding(self)) {
         self.position = 0
         @ self!READ (self.position) → (newPosition, text)
         self.position = newPosition
         textEditReadOnly(self, text)
     }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(documentStyle)
{
    self.grstyle = documentStyle
    self.position = 0
    self.reading = FALSE
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

A single static image of (unopened) document for each document style. Text or picture display of contents of document.

Host behavior:

```
GET
PUT
READ (position) → (newPosition, text)
{
    if (holding(self) && position < self.maximumLength) {
        text = content[position...position+READ_LENGTH]
        newPosition = position + READ_LENGTH
    } else {
        text = ""
        newPosition = 0
    }
}
THROW
```

Object:

boomerang

Description:

Your standard Australian throwing toy.

Function:

Joke.

Notes:

The boomerang is rather peculiar. It works like a ball or frisbee, except that when you throw it it disappears off the edge of the region. At some random time in the future (perhaps days or weeks later) it returns to you, wherever you are.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

```
returnTo        /* Avatar to return to at some random time in the future. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
{
    /* Throw away. */
    @ self!THROWAWAY () → ()
    /* *** Need to animate motion to edge of screen. *** */
    destroyObject(self)
}
```

Initialization:

```
nullInit()
```

Destruction:

```
smash()
```

Asynchronous actions:

```
THROWAWAY* ()
{
    /* *** Need to animate motion to edge of screen. *** */
    destroyObject(self)
}
```

Graphics:

Two or three state image of flying boomerang. One frame of this will suffice for boomerang at rest.

Host behavior:

```
BoomerangReturn()
{
    /* *** Make boomerang reappear in region where thrower is now located,
```

```
    and fly in and either get caught in the avatar's hands (if empty
    handed) or drop at it's feet. *** */
}
GET
PUT
THROWAWAY () → ()
{
    if (holding(self)) {
        self.returnTo = avatar
        # self → THROWAWAY* ()
        scheduleEvent(BoomerangReturn, randomTimeInTheFuture())
        disappearObject(self)
    }
}
```

Object:

bottle

Description:

A glass bottle.

Function:

Holds water, other liquids.

Notes:

The bottle can hold liquids. There is only one liquid substance defined right now, water, but we may add others as the design evolves (e.g., oil, fuel, etc.). The bottle is either filled or empty. There are no in-between states.

Styles:

None.

Properties:

```
filled          /* Flag telling whether bottle is filled or empty.  TRUE if
                   filled. */
contents        /* What is in the bottle (for now, this is always water). */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
{
```

/ Pour the contents of the bottle on the indicated spot. Of course, you can only do this with a full bottle. */*

```
    if (goTo()) {
        if (self.filled) {
            @ self!POUR () → ()
            self.filled = FALSE
            avatar.action = POUR
        }
    }
}
```

Initialization:

```
init(filled, contents)
{
    self.filled = filled
    self.contents = contents
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Two state image: bottle empty and bottle full.

Host behavior:

```
FILL () → (success)
{
    success = (holding(self) && !self.filled && atWater())
    if (success) {
        self.filled = TRUE
        self.contents = WATER_CLASS
        # avatar → FILL* ()
    }
}
GET
POUR () → ()
{
    if (holding(self) && self.filled) {
        self.filled = FALSE
        self.contents = NULL
        # avatar → POUR* ()
    }
}
PUT
```

Object:

box

Description:

Your basic box.

Function:

Can carry more than two items using only two hands.

Notes:

A box is simply an object carrier that can be carried in the hands. It is like a bag, but it has a larger capacity.

Styles:

```
BOX_CARDBOARD, BOX_CRATE, BOX_TREASURE_CHEST, BOX_PICNIC_BASKET
```

Properties:

```
contents        /* A list of the objects in the box. */
open            /* Flag that box is open.  TRUE if it is. */
key             /* Key to unlock box.  Always NULL. */
unlocked        /* Flag that box is unlocked.  Always TRUE. */
```

Class properties:

```
capacity = ?            /* How many things this can hold. */
displayContents = FALSE /* Whether we show what's in it. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
adjacentOpenCloseContainer()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndPickFromOrGet()
```

Put:

```
goToAndDropIntoIfOpen()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(boxStyle, open, contentsVector)
{
    self.grstyle = boxStyle
    self.open = open
    self.key = NULL
    self.unlocked = TRUE
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

Single static image of box for each style.

Host behavior:

```
CLOSECONTAINER
GET
```

```
OPENCONTAINER
PUT
THROW
```

Object:

bridge

Description:

Your basic small foot or highway bridge.

Function:

Provides a pathway across water.

Notes:

This is a large scenic background object. It is different from most objects in that the location on it that the player points to can be significant. Topologically, it is a parallelepiped on the ground. One pair of sides (the ‘‘ends’’) are of a fixed length (the ‘‘width’’) which will be the same for all bridges. The other pair of sides (the ‘‘sides’’) may vary in length depending on the length of the bridge. The length will be some integer multiple of of the standard bridge section length, which is in turn based on the size of the imagery. It is possible to walk around on top of the bridge’s surface and to cross the ends, but it is not possible to cross the sides.

Styles:

None.

Properties:

```
orientation     /* Which way the the bridge is oriented w.r.t. p.o.v.
                   Possible values are BRIDGE_SIDE and BRIDGE_FRONT */
width           /* The number of standard bridge sections wide it is. */
length          /* The number of standard bridge sections long it is. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goToCursor()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToCursorAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can’t happen */
```

Initialization:

```
init(orientation, width, length)
{
    self.orientation = orientation
    self.width = width
    self.length = length
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Two static images of a bridge section: side view and end view. Which is used depends on the value of `orientation`. These are replicated end-to-end (for the side view) or back-to-front (for the end

view) as many times as there are bridge sections.

Host behavior:

None.

Object:

building

Description:

Any sort of building.

Function:

A ‘‘glue’’ object to organize building parts into a single entity.

Notes:

This is a rather special sort of object. It is not a selectable object on the screen. Rather, it is a mechanism for the host to tell the home system about a whole complex of objects without having to detail them each individually. A building is a collection of building parts: roof, walls, doors, windows, etc., that are displayed together. It is used as a backdrop to a region to show that the region behind it is a building interior of some sort. Buildings are not described by the host in detail. Instead, the host simply provides a building-type descriptor and the initialization and drawing code for the building object provide the rest. This means that the spectrum of possible buildings is limited compared to the range of things possible in general given our graphics capabilities, but it also means that the communications overhead to describe a building is vastly reduced.

Styles:

None.

Properties:

```
descriptor      /* What sort of building this is.  The descriptor is three
                   or four bytes that tell how it is put together.  Various
                   bits encode such things as the roof style, the roof
                   pattern, the height (in stories) of the building, the
                   width (in standard building sections) of the building, what
                   each section is (wall, door, window, etc.), what sort of
                   doors it has, what sort of windows it has, what the wall
                   siding is and what color it is.  *** The exact encoding
                   is TBD. *** */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

None, since it is not a selectable object on the screen.

Initialization:

```
init(descriptor)
{
    createBuilding(descriptor)
}
```

Destruction:

None.

Asynchronous actions:

None.

Graphics:

The building is just the sum of its parts, produced algorithmically from the descriptor.

Host behavior:

None.

Object:

bus

Description:

Mass transit.

Function:

Carries many avatars and their possessions along roads.

Notes:

A bus is simulated by having a region whose location changes with time. Busses follow predetermined paths. You board a bus by entering its region when it is at an adjacent bus stop. You get off by leaving its region when it is stopped. It costs money to ride the bus. Bus fare is collected by a fare box object — each bus region contains one. The bulk of the player interface to the bus itself is through the fare box object. The bus itself is not represented by an explicit object in the home system.

Styles:

None.

Properties:

No properties: it is not an object.

Class properties:

No class properties: it is not an object.

Host properties:

```
schedule        /* The sequence of regions the bus will visit and the times
                   that it will move between them. */
```

Command Behavior:

None: a bus is just a special kind of region. It is not a selectable object on the screen.

Initialization:

None: it is not really an object.

Destruction:

None: it is not really an object.

Asynchronous actions:

None

Graphics:

None: it is not really an object.

Host behavior:

Code to run bus.

Object:

bush

Description:

Your basic bushy plant.

Function:

Scenic element. Obstruction.

Notes:

This is a fairly inert scenic element, provided almost entirely for visual appeal.

Styles:

`BUSH_LARGE, BUSH_SMALL, BUSH_TUMBLEWEED, BUSH_SHRUB`

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(bushStyle)
{
    self.grstyle = bushStyle
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

One static image for each style of bush.

Host behavior:

None.

Object:

car

Description:

The great American automobile.

Function:

Carries avatars and their possessions along roads.

Notes:

The car is a vehicle that can travel on roads. A car can not ordinarily travel on plain ground, unless it is one of the rare off-road vehicles.

Styles:

None.

Properties:

```
contents        /* A list of the objects in the car.  NULL if there are
                   none. */
inhabitants[CAR_SIZE] /* The folks riding in it.  There are four slots which
                   are NULL if unoccupied.  The avatar in slot 0 gets to
                   drive. */
moving          /* A flag that the car is in motion. */
targetX         /* Motion destination X position. */
targetY         /* Motion destination Y position. */
offRoadVehicle  /* Flag that this car can travel off the road on ordinary
                   ground. */
```

Class properties:

```
capacity = ?            /* How many things this can hold. */
displayContents = FALSE /* Whether to show what's in this. */
maxOccupants = CAR_SIZE = 4
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
climbInOrOut()
```

Stop:

```
cease()
```

Get:

```
goToAndPickFrom()
```

Put:

```
goToAndDropIn()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(moving, targetX, targetY, offRoadVehicle, inhabitants[CAR_SIZE], contentsVector)
{
    self.moving = moving
    self.targetX = targetX
    self.targetY = targetY
    self.offRoadVehicle = offRoadVehicle
    for (i=0; i<CAR_SIZE; ++i) {
        self.inhabitants[i] = inhabitants[i]
    }
    unpackContentsVector(self, contentsVector)
```

```
    }
```

Destruction:

```
    smashWithContents()
```

Asynchronous actions:

```
    DRIVE*
```

Graphics:

Three static images of car: side view, front view, back view. Sound of car engine driving.

Host behavior:

None.

Object:

chair

Description:

An ordinary chair.

Function:

Can be sat in.

Notes:

The chair is a type of scenic object, usually found in building interiors. Avatars can sit in ti. This doesn't really change anything, it's just for appearance.

Styles:

```
CHAIR_ARM, CHAIR_KITCHEN, CHAIR_STOOL, CHAIR_DESK
```

Properties:

```
orientation     /* How the chair is oriented w.r.t. p.o.v.  It can be any of
                   CHAIR_LEFT_SIDE, CHAIR_RIGHT_SIDE, CHAIR_FRONT or
                   CHAIR_BACK. */
occupants[CHAIR_SIZE]  /* Avatar that is sitting here.  NULL if nobody is. */
```

Class properties:

```
maxOccupants = CHAIR_SIZE = 1
```

Host properties:

No other properties.

Command Behavior:

Do:

```
sitOrGetUp()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(chairStyle, orientation, occupant)
{
    self.grstyle = chairStyle
    self.orientation = orientation
    self.occupants[0] = occupant
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Three static images for each style of chair: side view, front view and back view.

Host behavior:

None.

Object:

chest

Description:

A chest of drawers.

Function:

Can hold stuff.

Notes:

The chest of drawers is both a piece of furniture, for scenic purposes, and a container. It is a public container with two states, open and closed. When it is open, anyone in the region can see the contents, thus a chest of drawers may only be opened if there are enough free object identifiers available to identify the complete contents to all the players.

Styles:

None.

Properties:

```
contents        /* Contents list: a list of the objects that the chest
                   contains.  This will always be NULL if the chest is
                   closed. */
open            /* Flag telling whether the chest is open.  TRUE if open. */
key             /* Key to unlock this.  Always NULL since has no lock. */
unlocked        /* Flag that it is unlocked.  Always TRUE since no lock. */
```

Class properties:

```
capacity = ?            /* How many things this can hold. */
displayContents = FALSE /* Whether we should show what's in it. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
adjacentOpenCloseContainer()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndPickFromIfOpen()
```

Put:

```
goToAndDropIntoIfOpen()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(open, contentsVector)
{
    self.open = open
    self.key = NULL
    self.unlocked = FALSE
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

Two state image of chest, showing it open and closed.

Host behavior:

```
CLOSECONTAINER
OPENCONTAINER
```

Object:

club

Description:

The most basic weapon.

Function:

Pain and injury at close range.

Notes:

The club is the simplest weapon. You go up to somebody and you hit them with it.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

`depends()`

Go:

`goTo()`

Stop:

`cease()`

Get:

`goToAndGet()`

Put:

`goToAndDropAt()`

Talk:

`broadcast()`

Reversed Do:

`strike()`

Initialization:

`nullInit()`

Destruction:

`smash()`

Asynchronous actions:

None.

Graphics:

Single static image of club. Possibly multiple orientations (horizontal, vertical, maybe diagonal) to make the animation look right. Sound of club striking avatar. Sound of club striking object.

Host behavior:

`ATTACK`
`BASH`
`GET`
`PUT`

Object:

coke machine

Description:

Soda pop vending machine.

Function:

Decorative scenic object, joke.

Notes:

The coke machine mostly just sits there. If you put a Token in it it will take it, but it will never actually give you a coke.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

```
take            /* Number of tokens the machine has stolen from innocent
                   avatars. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
{
    /* Try to buy a coke (nothing will happen except the payment). */
    payCoinOpOrDrop(COKE_COST, SOUND_JINGLE, SOUND_CLICK_CLUNK,
        SOUND_JINK, goTo)
}
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
nullInit()
```

Destruction:

```
smallExplosion()
```

Asynchronous actions:

```
PAY* ()
{
    spend(COKE_COST)
    soundEffect(CLICK_CLUNK)
}
```

Graphics:

Single static image of coke machine sitting there.

Host behavior:

```
PAY () → (success)
{
    success = spend(COKE_COST)
    if (success) {
```

```
        # self → PAY* ()
    }
}
```

Object:

compass

Description:

Your basic pointer to the West Pole.

Function:

Tells absolute direction.

Notes:

The compass doesn't actually DO anything. It is simply displayed in a manner which indicates the direction that is West.

Styles:

None.

Properties:

```
orientation     /* The direction to the West pole, in the form of a number
                   from 0 to 3, where 0 indicates straight ahead from the
                   viewpoint, 1 indicates directly to the right, 2 indicates
                   directly towards the viewpoint, and 3 indicates directly to
                   the left. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(orientation)
{
    self.orientation = orientation
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Four state image of compass with arrow: pointing left, pointing right, pointing towards viewpoint, pointing away from viewpoint.

Host behavior:

```
GET
PUT
THROW
```

Object:

couch

Description:

Your basic living room couch.

Function:

Like chair, but can be sat in by multiple avatars.

Notes:

A couch works just like a chair, but it can hold two avatars. We have to maintain the position that each is sitting in, so we pay attention to where the player pointed to when he gave the command to sit, and sit in the open slot closest to that point.

Styles:

```
COUCH_STUFFED, COUCH_MODERN
```

Properties:

```
orientation     /* How the couch is oriented w.r.t. p.o.v.  Can be any of
                   COUCH_LEFT_SIDE, COUCH_RIGHT_SIDE, COUCH_FRONT or
                   COUCH_BACK. */
occupants[COUCH_SIZE] /* The folks sitting there.  There are two slots which
                   are NULL if unoccupied. */
```

Class properties:

```
maxOccupants = COUCH_SIZE = 2
```

Host properties:

No other properties.

Command Behavior:

Do:

```
sitOrGetUp()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(couchStyle, orientation, occupants[COUCH_SIZE])
{
    self.grstyle = couchStyle
    self.orientation = orientation
    for (i=0; i<COUCH_SIZE; ++i) {
        self.occupants[i] = occupants[i]
    }
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Three static images for each style of couch: side view, front view and back view.

Host behavior:

None.

Object:

countertop

Description:

Store counter.

Function:

Can support things. Mediates transactions.

Notes:

The countertop is a mechanism for mediating transactions. Objects placed on the counter follow a slightly different protocol when being picked up by passers by. If you place an object on the counter, you can pick it up again. However, if somebody else places an object on the counter, you can only pick it up if you already placed an item of your own on the counter. This interlock is mediated by the host, when it decides whether a GET operation has succeeded or not.

Styles:

None.

Properties:

```
contents        /* Contents list: a list of the objects that the countertop
                   contains. */
```

Class properties:

```
capacity = ?            /* How many things this can hold. */
displayContents = TRUE  /* Whether we should show what's in it. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropOnto()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(contentsVector)
{
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

Single static image of grocery store type countertop.

Host behavior:

None.

Object:

credit card

Description:

Your basic plastic money.

Function:

Can be used at participating businesses to pay for purchases. Don't leave home without it.

Notes:

The credit card operates exactly like a token, with two important exceptions: First, it debits directly from your bank account (so it's actually a debit card and not a credit card!). Second, someone cannot take it from you and use it like cash the way a token can.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
{
    if (target.class == AVATAR_CLASS) {
        amount = selectDenomination(bankAccountBalance)
        @ self!PAYTO (target.noid, amount) → (success)
        if (success) {
            bankAccountBalance -= amount
            avatar.action = PAY
        }
    }
}
```

Initialization:

```
nullInit()
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image of card.

Host behavior:

```
GET
PUT
PAYTO (targetId, amount) → (success)
```

```
{
    target = ↑targetId
    success = FALSE
    if (target.class == AVATAR_CLASS && amount >= avatar.bankAccountBalance) {
        avatar.bankAccountBalance -= amount
        target.bankAccountBalance += amount
        target → PAYTO* (amount)
        success = TRUE
    }
}
```

Object:

crystal ball

Description:

Your basic crystal ball.

Function:

Oracle and diviner.

Notes:

The crystal ball functions rather like the fountain, but on a more personal level. It answers questions, makes predictions, and generally does all of the things that oracles are supposed to do. However, it is, by tradition, subtle and not completely reliable. Sometimes it answers your questions immediately. Sometimes it even carries on a conversation with you. Most of the time though, it takes quite a while to get an answer: you have to come back several days later. Occasionally it says things spontaneously. Such things are usually important. Anyone in the region with the crystal ball can hear what it says. Of course, the minds behind the ball are our own. Somebody has to respond, but the nature of the oracle business is such that a response need not be timely, nor need all questions be answered.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

```
questions       /* List of questions asked of the crystal ball. */
answers         /* List of answers to questions waiting to be sent out.
                   *** When? *** */
live            /* Flag that the ball is 'live', i.e., a human is acting as
                   the voice of the crystal, rather than having it function
                   off-line. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
{
    /* Ask the crystal ball a question. */
    @ self!ASKORACLE (text) → ()
}
```

Reversed Do:

```
throw()
```

Initialization:

```
nullInit()
```

Destruction:

```
mediumExplosion()
```

Asynchronous actions:

```
ORACLESPEAK+
```

Graphics:

Single static image of crystal ball. Swirling effect when it is working.

Host behavior:

ASKORACLE
GET
PUT
THROW

Object:

die

Description:

A six-sided die.

Function:

For gambling, games.

Notes:

The die is a simple object, like the compass, whose only real function is to display its own state. Graphically, this probably won't work. These things really want to travel in pairs.

Styles:

None.

Properties:

```
state           /* A number from 1 to 6 telling the die's face value. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    /* If the die is in hand, roll it to get a new random state value from
    the host. */
    if (holding(self)) {
        @ self!ROLL () → (result)
        self.state = result
    } else if (elsewhere(self)) {
        depends()
    }
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(state)
{
    self.state = state
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
ROLL* (newState)
{
    /* Asynchronous function called when somebody else rolls the die. */
    self.state = newState
}
```

Graphics:

6-state image showing the various possible die rolls.

Host behavior:

```
GET
PUT
ROLL () → (result)
{
    result = random() % 6
    # self → ROLL* (result)
}
THROW
```

Object:

display case

Description:

Store display case.

Function:

Can hold things visibly but safely, even if unattended.

Notes:

The display case is a transaction mediator similar to the countertop. The protocol that it obeys is that objects placed in it may only be picked up again by the avatar who owns the display case. Thus, objects can be left open to view with impunity, even if unattended.

Styles:

None.

Properties:

```
contents        /* Contents list: a list of the objects that the display case
                   contains. */
owner           /* The avatar who owns this display case. */
```

Class properties:

```
capacity = ?            /* How many things this can contain. */
displayContents = TRUE  /* Whether to show what's in it. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropOnto()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(contentsVector)
{
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

Single static image of display case.

Host behavior:

None.

Object:

door

Description:

The conventional door.

Function:

Graphic element in buildings. Passageway through wall.

Notes:

A door is like a wall section, except that it can be opened to allow passage through it. You can only open a door if it is unlocked or if you have the key. We may want to encode the open/closed state and the locked/unlocked state into a single byte, to save room.

Styles:

```
DOOR_WOOD, DOOR_SCREEN, DOOR_STEEL
```

Properties:

```
open                    /* Flag indicating whether the door is open or closed.
                           TRUE if open. */
unlocked                /* Flag indicating whether or not the door is locked.
                           TRUE if unlocked. */
key                     /* Two-byte number indicating the key required for
                           this door. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
adjacentOpenClose()
```

Go:

```
goToOrPassThrough()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(doorStyle, open, unlocked, key)
{
    self.grstyle = doorStyle
    self.open = open
    self.unlocked = unlocked
    self.key = key
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

A two state image set (open door and closed door) for each door style.

Host behavior:

```
CLOSE
```

OPEN

Object:

dropbox

Description:

A convenient roadside mail drop box.

Function:

Interface to mail system.

Notes:

This is a variant on the standard mailbox. It is like a regular mailbox, except that it is send-only and anyone at all can use it. Also, of course, it *looks* like a dropbox!

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
{
    sendMail(NO_ADDRESS_CHECK)
}
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
nullInit()
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image of drop box just sitting there.

Host behavior:

SENDMAIL

Object:

drugs

Description:

Little pills.

Function:

Temporarily changes an avatar's properties.

Notes:

A drug object is represented as a pill bottle. The bottle contains a certain number of pills. This number is decremented each time you take one. When they run out the bottle disappears. Taking a pill causes something to happen to the avatar. There are a variety of possible effects, most of them bad. ***** The set of possible effects needs thought ***.**

Styles:

None.

Properties:

```
count           /* How many pills are left. */
```

Class properties:

No class properties.

Host properties:

```
effect          /* What the pills do.  Used as case switch for execution of
                   pills' behavior. */
```

Command Behavior:

Do:

```
{
    /* Take a pill. */
    if (holding(self)) {
        @ self!TAKE () → ()     /* Effect is conveyed asynchronously */
        self.count--
        avatar.action = TAKE
        if (self.count == 0) {
            destroyObject(self)
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(count)
{
    self.count = count
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image of pill bottle.

Host behavior:

```
GET
PUT
TAKE () → ()
{
    if (holding(self)) {
        self.count--
        (drugEffects[self.effect])()  /* *** define these *** */
        (*self.effect)()
        if (self.count == 0) {
            destroyObject(self)
        }
        # avatar → TAKE* ()
    }
}
THROW
```

Object:

escape device

Description:

Your basic panic button.

Function:

Gets you out of a jam, fast.

Notes:

This device is very rare and expensive. It is a little box with a button on it. If you press the button, you are instantly teleported back to your turf. Since this is such a powerful device, it is given only a limited charge. Each teleport uses one unit of charge. When the charge level reaches zero, the device disappears. Note that the typical charge found in practice should be 1.

Styles:

None.

Properties:

```
charge          /* How many teleports are left in this device before it runs
                   out. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    /* If in hand, press button and bug out.  Actual teleport will occur
    asynchronously. */
    if (holding(self)) {
        @ self!BUGOUT () → (success)
        if (success) {
            soundEffect(SOUND_ZAP)
            self.charge--
            if (self.charge == 0) {
                destroyObject(self)
            }
        }
    } else if (elsewhere(self)) {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(charge)
{
    self.charge = charge
```

```
    }
```

Destruction:

```
    smallExplosion()
```

Asynchronous actions:

None.

Graphics:

Single static image of device. Sound effect when device is activated.

Host behavior:

```
    BUGOUT () → (success)
    {
        if (holding(self)) {
            goToNewRegion(avatar, avatar.turf)
            self.charge--
            if (self.charge == 0) {
                destroyObject(self)
            }
            success = TRUE
            # avatar → BUGOUT* ()
        } else {
            success = FALSE
        }
    }
    GET
    PUT
    THROW
```

Object:

fake gun

Description:

A pistol (almost).

Function:

Apparant death and destruction from a distance, but not really.

Notes:

The fake gun works like the real gun, except when you shoot with it a flag that says "BANG!" comes out instead of actually shooting somebody.

Styles:

None.

Properties:

```
state           /* What state the gun is in.  Possible values are
                   FAKE_GUN_READY and FAKE_GUN_FIRED. */
safetyOn        /* Flag that the safety is on.  TRUE if it is. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
```

/ Toggle the safety. Note: the safety operation is strictly local. It is only to prevent you from accidently shooting somebody, and is thus part of the "user interface" rather than an intrinsic part of the operation of the object. If the flag is out, push it back in. */*

```
    if (holding(self)) {
        if (self.state == FAKE_GUN_FIRED) {
            self.state = FAKE_GUN_READY
            @ self!RESET () → ()
        } else if (self.safetyOn) {
            soundEffect(SOUND_SAFETY_OFF)
            self.safetyOn = FALSE
        } else {
            soundEffect(SOUND_SAFETY_ON)
            self.safetyOn = TRUE
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
{
```

/ Try to attack somebody or something with fake gun. */*

```
        if (!self.safetyOn && self.state == FAKE_GUN_READY) {
            soundEffect(SOUND_BANG)
            avatar.action = SHOOT
            self.state = FAKE_GUN_FIRED
            @ self!FAKESHOOT () → ()
        } else {
            soundEffect(SOUND_DUD)
        }
    }
```

Initialization:

```
    init(state, safetyOn)
    {
        self.state = state
        self.safetyOn = safetyOn
    }
```

Destruction:

```
    smash()
```

Asynchronous actions:

```
    RESET* ()
    {
        self.state = FAKE_GUN_READY
    }
    FAKESHOOT* ()
    {
        soundEffect(SOUND_BANG)
        self.state = FAKE_GUN_FIRED
    }
```

Graphics:

Images of static gun and gun with FLAG in front and side views. Sound of shot.

Host behavior:

```
    FAKESHOOT () → ()
    {
        if (holding(self) && self.state == FAKE_GUN_READY) {
            self.state = FAKE_GUN_FIRED
            # self → FAKESHOOT* ()
        }
    }
    GET
    RESET () → ()
    {
        if (holding(self) && self.state == FAKE_GUN_FIRED) {
            self.state = FAKE_GUN_READY
            # self → RESET* ()
        }
    }
    PUT
```

Object:

fare box

Description:

Mass transit fare collection box.

Function:

Controls access to the bus system.

Notes:

The bus system is based on the region model. A bus is simply a region whose connectivity with the rest of the world changes with time. You get on the bus by entering the bus region, and leave the bus by exiting it, just as you would any other region. The bus follows a bus route, and every few minutes it moves from one stop to another. If you are on it when it moves, you move with it. However, to ride the bus you must pay bus fare. Bus fare is set by the **MicroCosm** authorities and is uniform throughout the world. If you do not pay, you will be automagically transported out of the bus region when the bus leaves and deposited in the bus stop region you entered from.

Styles:

None.

Properties:

```
nextStop        /* String with name of next stop on bus line */
```

Class properties:

No class properties.

Host properties:

```
take            /* How much money this box has collected in bus fares. */
```

Command Behavior:

Do:

```
{
    /* Announce next stop */
    balloonMessage(self, "Next stop %s", self.nextStop)
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
{
    /* If item in hand is a token and you have enough money for bus fare,
    drop amount of fare in fare box, otherwise drop at foot of box. */
    payCoinOpOrDrop(BUS_FARE, NULL, SOUND_DEPOSIT, SOUND_REJECT, goto)
}
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(nextStop)
{
    self.nextStop = nextStop
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
PAY* ()
{
```

```
        spend(BUS_FARE)
        soundEffect(SOUND_DEPOSIT)
    }
    DEPARTING+ (time)
    {
        /* Asynchronous message from host a few minutes before departure */
        /* *** Trigger this. *** */
        balloonMessage(self, "Departing for %s in %d minutes", self.nextStop,time)
        soundEffect(SOUND_ANNOUNCE)
    }
    DEPARTURE+ ()
    {
        /* Asynchronous message from host upon departure */
        /* *** Trigger this. *** */
        balloonMessage(self, "Departing now for %s", self.nextStop)
        soundEffect(SOUND_ANNOUNCE)
    }
    ARRIVAL+ (newNextStop)
    {
        /* Asynchronous message from host upon arrival */
        /* *** Trigger this. *** */
        balloonMessage(self, "Arriving at %s", self.nextStop)
        soundEffect(SOUND_ANNOUNCE)
        self.nextStop = newNextStop
        balloonMessage(self, "Next stop %s", self.nextStop)
    }
```

Graphics:

Single static image of fare box. Sound of coin dropping in box. Sound of rejection for insufficient fare. Sound announcing bus departure or arrival.

Host behavior:

```
    PAY () → (success)
    {
        if (spend(BUS_FARE)) {
            success = TRUE
            avatar.busFarePaid = TRUE
            self.take++
            # self → PAY* ()
        } else {
            success = FALSE
        }
    }
```

Object:

fence

Description:

A section of an impassable, man-made barrier.

Function:

Linear obstruction.

Notes:

Each fence object is a single section of fence, running vertically, horizontally, or diagonally with respect to the viewpoint. Fence sections are of a standard size (say 10 feet long), so the path of a fence section can be reckoned by the (x, y) location of one end together with a direction indicator. A fence blocks passage across the line that it runs on. For visual stylistic reasons, there are several different styles of fencing available.

Styles:

```
FENCE_CYCLONE, FENCE_PICKET, FENCE_WOOD, FENCE_STONE, FENCE_BRICK
```

Properties:

```
direction       /* Which direction the fence runs from the location point, in
                   the form of a number from 0 to 8, where 0 indicates WEST, 1
                   indicates NORTHWEST, 2 indicates NORTH, and so on clockwise
                   around the compass rose.  WEST is always taken as the
                   direction directly away from the viewpoint, even if this
                   is not true West. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(fenceStyle, direction)
{
    self.grstyle = fenceStyle
    self.direction = direction
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Three static images of a fence section: side view, end view and diagonal view, for each style of fence. May be composed entirely of cels, or a combination of cels and texture-mapped trapezoids. In the latter case, cels would be used for the fence posts and trapezoids for the fence walls proper. (Note:

using trapezoids would allow us to express fence sections in terms of two arbitrary endpoints instead of a single end and a direction vector).

Host behavior:

None.

Object:

flag

Description:

The colors.

Function:

Scenic decoration.

Notes:

A flag is a decoration as well as a marker that can be carried around.

Styles:

None.

Properties:

```
state           /* What state the flag is in.  Possible values are
                   FLAG_WAVING, FLAG_FURLED and FLAG_EXTENDED. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends();
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(state)
{
    self.state = state
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
CHANGESTATE+ (newState)
{
    /* Asynchronous message from host when the wind changes. */
    /* *** Trigger this. *** */
    self.state = newState
}
```

Graphics:

Animation of flag waving. Static image of flag furled. Static image of flag extended.

Host behavior:

```
GET
PUT
THROW
```

Object:

flashlight

Description:

Your basic hand torch.

Function:

Portable light source at night.

Notes:

The **MicroCosm** follows a cycle of day and night. When it is night-time, the screen is displayed in darkened colors, which makes things hard to see. Having a flashlight in the region makes the region appear like daytime (provided that the light is turned on, of course), even if it is night elsewhere. We may wish to consider requiring batteries, but for now it just works forever.

Styles:

None.

Properties:

```
on              /* Flag telling whether the light is on or off.  TRUE if it is
                   on. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    /* If holding, turn on or off.  Otherwise, depends. */
    if (toggleSwitch(holding(self))) {
        if (self.on) {
            lightLevel++
        } else {
            lightLevel--
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(on)
{
    self.on = on
}
```

Destruction:

```
smash()
```

Asynchronous actions:

OFFLIGHT*
ONLIGHT*

Graphics:

Static image of flashlight in two orientations: horizontal and vertical. Which orientation to use depends on how it is being held (***** unsolved problem*****).

Host behavior:

GET
OFF
ON
PUT
THROW

Object:

floor lamp

Description:

A household floor lamp.

Function:

Provides light at night.

Notes:

The **MicroCosm** follows a cycle of day and night. When it is night-time, the screen is displayed in darkened colors, which makes things hard to see. Having a floor lamp in the region makes the region appear like daytime (provided that the lamp is turned on, of course), even if it is night elsewhere. The floor lamp is a decorative furniture object also.

Styles:

None.

Properties:

```
on              /* Flag telling whether or not the lamp is turned on.  TRUE if
                   it is on. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    /* Flip the light switch. */
    if (toggleSwitch(!elsewhere(self))) {
        if (self.on) {
            lightLevel++
        } else {
            lightLevel--
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(on)
{
    self.on = on
}
```

Destruction:

```
smash()
```

Asynchronous actions:

`OFFLIGHT*`

`ONLIGHT*`

Graphics:

Single static image of floor lamp.

Host behavior:

`OFF`

`ON`

Object:

fortune machine

Description:

A vending machine that tells your fortune.

Function:

Humor.

Notes:

The fortune machine is a little like the oracle, but less profound. You give it a token and it tells you a fortune. The fortunes are little proverbs, sayings, predictions and pieces of advice in the tradition of fortune cookie programs everywhere.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

```
count            /* How many fortunes this machine has dispensed. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
{
     /* Spend the price and speak the saying that comes out. */
     if (payCoinOpOrDrop(FORTUNE_COST, SOUND_JINGLE,
               SOUND_CLICK_CLUNK, SOUND_JINK, goTo)) {
          balloonMessage(self, "%s", text)
     }
}
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
nullInit()
```

Destruction:

```
smash()
```

Asynchronous actions:

```
SPEAKFORTUNE* (text)
{
     /* Asynchronous message from host when somebody else puts coin in
     machine. */
     balloonMessage(self, "%s", text)
}
```

Graphics:

Single static image of fortune machine.

Host behavior:

```
PAY () → (success, text)
```

```
{
    if (spend(FORTUNE_COST)) {
        success = TRUE
        text = fortunes[random() % FORTUNE_COUNT]
        # self → SPEAKFORTUNE* (text)
    } else {
        success = FALSE
        text = ""
    }
}
```

Object:

fountain

Description:

Generic looking tacky town square fountain.

Function:

Scenic element. Water source. Oracle.

Notes:

The fountain is a major scenic element, but its most dramatic function is as oracle. The oracle answers questions, makes predictions, and generally does all of the things that oracles are supposed to do. However, it is, by tradition, subtle and not completely reliable. Sometimes it answers your questions immediately. Sometimes it even carries on a conversation with you. Most of the time though, it takes quite a while to get an answer: you have to come back several days later. Occasionally it says things spontaneously. Such things are usually important. Anyone in the region with the oracle can hear what it says. Of course, the minds behind the oracle are our own. Somebody has to respond, but the nature of the oracle business is such that a response need not be timely, nor need all questions be answered.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

```
questions       /* List of questions asked of the oracle. */
answers         /* List of answers to questions waiting to be sent out.
                   *** When? *** */
live            /* Flag that the oracle is 'live', i.e., a human is acting as
                   the voice of the oracle, rather than having it function
                   off-line. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndFill()
```

Put:

```
goToAndDropAt()
```

Talk:

```
{
    /* Ask the oracle a question. */
    @ self!ASKORACLE (text) → ()
}
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
nullInit()
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

```
ORACLESPEAK+
```

Graphics:

Single static image of fountain. Possible animation of water spurting up. Possible process sound effect of water spurting.

Host behavior:

ASKORACLE

Object:

frisbee

Description:

Your standard disk throwing toy.

Function:

For playing games.

Notes:

The frisbee is relatively inert. You can pick it up and throw it. If you throw it at an avatar who is not already holding something else, that avatar will catch it. The frisbee operates just like the ball, except that it has a different sort of flight path.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

`depends()`

Go:

`goTo()`

Stop:

`cease()`

Get:

`goToAndGet()`

Put:

`goToAndDropAt()`

Talk:

`broadcast()`

Reversed Do:

`throw()` **/* *** Need something special here. *** */**

Initialization:

`nullInit()`

Destruction:

`smash()`

Asynchronous actions:

None.

Graphics:

Two-state (orientation dependent) image of frisbee: flat on and edge on.

Host behavior:

`GET`
`PUT`
`THROW`

Object:

garbage can

Description:

Conventional garbage can or wastebasket.

Function:

Makes things disappear.

Notes:

The garbage can is a means for getting rid of things. It operates like any other container, except that upon command it will cause to disappear anything inside it.

Styles:

```
GARBAGE_CAN, GARBAGE_WASTEBASKET
```

Properties:

```
contents        /* Contents list: a list of the objects that the garbage can
                   contains.  */
```

Class properties:

```
capacity = ?            /* How many things this holds. */
displayContents = FALSE /* Whether we should show what's in it. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    /* Empty the trash. */
    if (!elsewhere(self)) {
        @ self!FLUSH () → ()
        purgeContents(self)
        soundEffect(SOUND_FLUSH)
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndPickFrom()
```

Put:

```
goToAndDropInto()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(garbageCanStyle, contentsVector)
{
    self.grstyle = garbageCanStyle
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

```
FLUSH* ()
{
```

```
        purgeContents(self)
        soundEffect(SOUND_FLUSH)
    }
```

Graphics:

A single static image of the garbage can for each style. Sound effect of garbage being flushed.

Host behavior:

```
    FLUSH () → ()
    {
        if (!elsewhere(self)) {
            destroyContents(self)
            # self → FLUSH* ()
        }
    }
```

Object:

gate

Description:

A gate in a fence or wall.

Function:

Provides a way to put a passageway through a fence.

Notes:

A gate is like a fence section, except that it can be opened to allow passage through it. You can only open a gate if it is unlocked or if you have the key. We may want to encode the open/closed state and the locked/unlocked state into a single byte, so that we can use a simple table to determine what to do when opening and closing, rather than having complicated if-then-else code.

Styles:

```
FENCE_CYCLONE, FENCE_PICKET, FENCE_WOOD, FENCE_STONE, FENCE_BRICK
```

Properties:

```
open                    /* Flag indicating whether the gate is open or closed.
                           TRUE if open. */
unlocked                /* Flag indicating whether or not the gate is locked.
                           TRUE if unlocked. */
key                     /* Two-byte number indicating the key required for
                           this gate. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
adjacentOpenClose()
```

Go:

```
goToOrPassThrough()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(gateStyle, open, unlocked, key)
{
    self.grstyle = gateStyle
    self.open = open
    self.unlocked = unlocked
    self.key = key
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

A two state image set (open gate and closed gate) for each fence style.

Host behavior:

```
CLOSE
OPEN
```

Object:

gemstone

Description:

Like a rock, only worth more.

Function:

Valuable. May contain magic.

Notes:

Some objects in the **MicroCosm** are magical. A magical object is one that does something special and powerful in addition to whatever other things the object would do intrinsically. The magical function may vary. ***** The set of magical functions needs thought. *****

Styles:

```
GEM_RUBY, GEM_DIAMOND, etc.
```

Properties:

```
magic           /* Flag that this is a magical gemstone.  (Perhaps we should
                   keep this a secret?). */
```

Class properties:

No class properties.

Host properties:

```
magicType       /* What sort of magic this gemstone contains.  Used as case
                   switch for execution of magical behavior. */
```

Command Behavior:

Do:

```
doMagicIfMagic()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(gemstoneStyle, magic)
{
    self.grstyle = gemstoneStyle
    self.magic = magic
}
```

Destruction:

```
smallExplosionIfMagic()
```

Asynchronous actions:

Functions appropriate for home-resident part of magical behavior.

Graphics:

Single static image of gemstone. Stylistic variations possible due to stone style.

Host behavior:

```
GET
MAGIC
PUT
THROW
```

Object:

generic flat animal

Description:

Anonymous mammal that was run over by a truck sometime earlier.

Function:

Joke.

Notes:

This is a completely inert scenic object.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

`depends()`

Go:

`goTo()`

Stop:

`cease()`

Get:

`goToAndGet()`

Put:

`goToAndDropAt()`

Talk:

`broadcast()`

Reversed Do:

`throw()`

Initialization:

`nullInit()`

Destruction:

`smash()`

Asynchronous actions:

None.

Graphics:

Single static image of animal laying in the road.

Host behavior:

`GET`
`PUT`
`THROW`

Object:

grenade

Description:

A little bomb that goes boom.

Function:

Death and mayhem in quantity.

Notes:

A grenade has a pin. It is harmless unless the pin is pulled. If the pin is pulled, then it will explode the next time it leaves the avatar's hand, blowing up everything at or near the destination location. Also, once the pin is pulled it may not be reinserted.

Styles:

None.

Properties:

```
pinPulled      /* Flag that the pin has been pulled.  TRUE if it has been. */
```

Class properties:

No class properties.

Host properties:

```
timer           /* Delay timer after pin is pulled. */
```

Command Behavior:

Do:

```
{
    /* Pull the pin. */
    if (holding(self) && !self.pinPulled) {
        @ self!PULLPIN () → (success)
        if (success) {
            self.pinPulled = TRUE
            soundEffect(SOUND_PINPULL)
            avatar.action = PULL_PIN
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw() /* Damage effects occur asynchronously. */
```

Initialization:

```
init(pinPulled)
{
    self.pinPulled = pinPulled
}
```

Destruction:

```
mediumExplosion()
```

Asynchronous actions:

```
EXPLODE+ ()
```

```
{
    /* Asynchronous message when grenade blows up. */
    explosionAt(self.x, self.y, SMALL_EXPLOSION)
    destroyObject(self)
    /* *** Damage? *** */
}
```

Graphics:

Two state image of grenade: static state and pin pulled. Animation of explosion. Sound of pin being pulled. Sound of explosion.

Host behavior:

```
GET
PULLPIN () → (success)
{
    if (holding(self) && !self.pinPulled) {
        self.pinpulled = TRUE
        success = TRUE
        scheduleEvent(GrenadeExplosion, GRENADE_FUSE_DELAY)
    } else {
        success = FALSE
    }
}
PUT
THROW
GrenadeExplosion()
{
    # self → EXPLODE+ ()
    self → EXPLODE+ ()
    destroyObject(self)
    /* *** Damage? *** */
}
```

Object:

ground

Description:

The basic background below the horizon.

Function:

Can be walked on. Can be pointed at, returning a location.

Notes:

This is not really an object, but is treated as one to make the player interface clean.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goToCursor()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToCursorAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
nullInit()
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

None.

Graphics:

Absolute blank, single color background from bottom of graphics window up to the horizon line (i.e., a filled rectangle). Drawn by background rendering initialization routines.

Host behavior:

None.

Object:

gun

Description:

A pistol.

Function:

Death and destruction from a distance.

Notes:

A gun works pretty much as you would expect. It does have a safety switch, which must unset before the gun will shoot. You shoot by pointing at the target and selecting **do**. We should decide if requiring ammunition would be a good idea.

Styles:

None.

Properties:

```
safetyOn        /* Flag that the safety is on.  TRUE if it is. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    /* Toggle the safety. Note: the safety operation is strictly local.  It
    is only to prevent you from accidently shooting somebody, and is thus
    part of the "user interface" rather than an intrinsic part of the
    operation of the object. */
     if (holding(self)) {
         if (self.safetyOn) {
             soundEffect(SOUND_SAFETY_OFF)
             self.safetyOn = FALSE
         } else {
             soundEffect(SOUND_SAFETY_ON)
             self.safetyOn = TRUE
         }
     } else {
         depends()
     }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
shoot()
```

Initialization:

```
init(safetyOn)
{
    self.safetyOn = safetyOn
}
```

Destruction:

`mediumExplosion()`

Asynchronous actions:

None.

Graphics:

Images of static gun and gun shooting in front and side views. Sound of shot. Sound of shot striking avatar. Sound of shot striking object.

Host behavior:

`ATTACK`
`BASH`
`GET`
`PUT`

Object:

hand of god

Description:

The finger of doom.

Function:

An indispensible system management tool.

Notes:

The hand of god is a giant animated hand that comes down off the top of the screen and fires a lightning bolt off its finger at a target, resulting a large explosion that leaves a small pile of smoking cinders in its wake.

Styles:

None.

Properties:

```
state           /* What state the hand is in.  Possible values are
                   GOD_DORMANT, GOD_FIRING and GOD_CINDERS. */
target          /* Who or what is to be blasted. */
```

Class properties:

No class properties.

Host properties:

No additional properties.

Command Behavior:

None: it is never selectable on the screen.

Initialization:

```
init(state, target)
{
    self.state = state
    self.target = target
}
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

```
BLAST* (target)
{
    self.target = target
    self.state = GOD_FIRING
    destroyObject(target)
}
```

Graphics:

Animation of hand, lightning bolt and cinder cone, as described above.

Host behavior:

Triggered on command by system operator.

Object:

hat

Description:

Your basic hat.

Function:

Decorative. Helps distinguish one avatar from another.

Notes:

A hat is simply a piece of clothing that serves to personalize an avatar. It is represented by a cel that simply follows the avatar's head.

Styles:

`HAT_BASEBALL, HAT_COWBOY, HAT_PANAMA, HAT_FEDORA, HAT_BERET, HAT_CRASH_HELMET, HAT_COMBAT_HELMET, HAT_SKI_MASK, etc.`

Properties:

No properties.

Class properties:

```
location = HEAD         /* Where this is worn. */
```

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
wear()
```

Put:

```
remove()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(hatStyle)
{
    self.grstyle = hatStyle
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Cel of hat for each style of hat in side, front and back views.

Host behavior:

```
GET
PUT
THROW
```

Object:

house cat

Description:

A lazy cat that lays around the house.

Function:

Scenic element. Puzzle with no solution.

Notes:

The housecat just lays there and sleeps. Every time to reenter the room its in a different spot. You can't pick it up or move it.

Styles:

None.

Properties:

No properties

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
nullInit()
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

A static image of a sleeping cat

Host behavior:

/* * Need to move the cat around when the region it is in is empty.**
Also, how does the cat get there in the first place? (Maybe it just shows up
and adopts you?) * */**

Object:

hot tub

Description:

A touch of Marin.

Function:

A place to hang out.

Notes:

The hot tub is a silly in-joke. In spite of its esoteric appearance, it actually operates more or less like a couch. The only difference is that it has a front part and a back part so that the avatars can be sandwiched in between and thus look like they are in the water. It's also a water source.

Styles:

None.

Properties:

```
occupants[TUB_SIZE] /* The folks sitting in it.  There are four slots which
                       are NULL if unoccupied. */
```

Class properties:

```
maxOccupants = TUB_SIZE = 4
```

Host properties:

No other properties.

Command Behavior:

Do:

```
sitOrGetUp()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndFill()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(occupants[TUB_SIZE])
{
    for (i=0; i<TUB_SIZE; ++i) {
        self.occupants[i] = occupants[i]
    }
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image of hot tub in two parts, front and back. We paint the back, then the avatars, then the front.

Host behavior:

None.

Object:

instant object pill

Description:

A little pill, until you use it.

Function:

Add water, it turns into some object.

Notes:

A instant object pill looks like an ordinary pill, until you pour water on it, at which point it transforms into some other sort of object. A instant object pill can, conceivably, turn into anything.

Styles:

None.

Properties:

```
instantWhat     /* Class of object that this turns into. */
```

Class properties:

No class properties.

Host properties:

```
initialization  /* Init vector. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(instantWhat)
{
    self.instantWhat = instantWhat
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
TRANSFORM* (initializationVector)
{
    /* Asynchronous action when somebody pours water on the pill. */
    /* *** Need to trigger this with pour. *** */
    newObject = createObject(self.instantWhat, self.container, self.x, self.y)
    soundEffect(SOUND_TRANSFORMATION)
    destroyObject(self)
    newObject.init(initializationVector)
}
```

Graphics:

Single static image of pill. Sound of metamorphosis.

Host behavior:

```
GET
PUT
```

THROW

Object:

jacket

Description:

Your basic jacket.

Function:

Decorative. Helps distinguish one avatar from another.

Notes:

A jacket is a clothing object whose primary purpose is to personalize avatars. It also has pockets, which are useful for holding things without resorting to hands. There is only one sort of jacket, but it comes in a variety of patterns.

Styles:

None.

Properties:

```
pattern         /* The pattern the jacket cels should be painted with. */
contents        /* A list of the objects in the jacket's pockets.  NULL if
                   the pockets are empty. */
```

Class properties:

```
location = TORSO        /* Where this is worn. */
capacity = ?            /* How many things this holds. */
displayContents = FALSE /* Whether to show what's in this. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
wearOrUnpocket()
```

Put:

```
remove()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(pattern, contentsVector)
{
    self.pattern = pattern
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

Single static image of jacket in heap, plus avatar animation cels showing jacket on torso in front, side and back views.

Host behavior:

```
GET
PUT
THROW
```

Object:

jukebox

Description:

The all-American Rockola.

Function:

Plays music.

Notes:

The jukebox is a coin operated music machine. It works more or less like a real jukebox. You drop a Token into it, press ‘‘B17’’ or something like that, and it starts playing music.

Styles:

None.

Properties:

```
playsToGo       /* How many plays the jukebox has left before the money runs
                   out. */
playing         /* Flag that jukebox is currently playing music (the jukebox
                   lights up when it is playing. */
```

Class properties:

No class properties.

Host properties:

```
music           /* The selection of music in the jukebox. */
musicCount      /* How many items are in the music selection list. */
catalogPtr      /* Current position flipping through the catalog. */
playlist        /* Selections queued for playing. */
playPointer     /* Where we are in the current selection. */
```

Command Behavior:

Do:

```
{
    /* Flip through music catalog. */
    if (goTo()) {
        @ self!CATALOG () → (letter, number, text)
        balloonMessage(self, "%c%d: %s", letter, number, text)
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
{
    /* If the avatar has a token in hand, activate the jukebox.  Otherwise,
    drop whatever is in hand next to the jukebox. */
    if (payCoinOpOrDrop(JUKEBOX_COST, SOUND_JINGLE,
            SOUND_CLICK_CLUNK, SOUND_JINK, goTo)) {
        self.playsToGo++
    }
}
```

Talk:

```
{
    /* Take the text string as the jukebox selection desired. */
    if (self.playsToGo > 0) {
        @ self!SELECT (text) → (success)
        if (success) {
```

```
                self.playsToGo--
            }
        } else {
            broadcast()
        }
    }
```

Reversed Do:

```
    /* This can't happen */
```

Initialization:

```
    init(playsToGo, playing)
    {
        self.playsToGo = playsToGo
        self.playing = playing
    }
```

Destruction:

```
    smash()
```

Asynchronous actions:

```
    PLAY+ (score)
    {
```

/ Asynchronous function when the host tells the box to play some tunes. The host actually sends the notes to play and our music driver plays them. */*

```
        /*** Trigger this. ***/
        self.playing = TRUE
        playMusic(score)
    }
```

Graphics:

Two state image of jukebox from the front: playing and not playing. When playing the jukebox is all lit up. When it's not it's not.

Host behavior:

```
    CATALOG () → (letter, number, text)
    {
        if (!elsewhere(self)) {
            letter = music[catalogPtr]->letter
            number = music[catalogPtr]->number
            text = music[catalogPtr]->text
            catalogPtr = (catalogPtr + 1) % musicCount
        }
    }
    PAY () → (success)
    {
        if (spend(JUKEBOX_COST)) {
            playsToGo++
            success = TRUE
        } else {
            success = FALSE
        }
    }
    SELECT (text) → (success)
    {
        selection = lookupSelection(music, musicCount, text)
        if (selection != NULL) {
            playsToGo--
            newPlay = alloc(PLAY_ENTRY)
```

```
            newPlay->play = selection
            newPlay->nextPlay = playList
            playList = newPlay
            if (!playing) {
                playing = TRUE
                playPointer = playList
            }
            success = TRUE
        } else {
            success = FALSE
        }
    }
```

Object:

key

Description:

A key for our unpickable locks.

Function:

Opens locked doors, containers.

Notes:

Each key object has a key number, as does each object with a lock (doors, containers, and so on). You can open locks with a key whose number matches that of the lock. It is possible for there to be more than one key object in the world with given key number, so that a lock can have multiple keys. To use a key, all you have to do is have it in your hand when you try to open the locked thing.

Styles:

None.

Properties:

```
keyNumber       /* What locks this key fits. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(keyNumber)
{
    self.keyNumber = keyNumber
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image of key.

Host behavior:

```
GET
PUT
THROW
```

Object:

knick knack

Description:

Your basic gewgaw.

Function:

Scenic element. Possibly magical.

Notes:

Knick knacks are usually inert scenic objects, intended only to clutter up a scene for purposes of visual variety. Occasionally, a knick-knack will be magical. The set of possible gewgaws is nearly infinite. We will pick a reasonable sample of two to four different styles.

Styles:

```
GEWGAW_TROPHY, GEWGAW_STATUETTE, GEWGAW_PAPERWEIGHT, GEWGAW_CANDELABRA,
GEWGAW_VASE
```

Properties:

```
magic           /* Flag that this is a magical knick-knack.  (Should we keep
                   this secret?) */
```

Class properties:

No class properties.

Host properties:

```
magicType       /* What sort of magic this gew-gaw contains.  Used as case
                   switch for execution of magical behavior. */
```

Command Behavior:

Do:

```
doMagicIfMagic()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(knickKnackStyle, magic)
{
    self.grstyle = knickKnackStyle
    self.magic = magic
}
```

Destruction:

```
smallExplosionIfMagic()
```

Asynchronous actions:

None.

Graphics:

A static image of the knick-knack for each style that there is.

Host behavior:

```
GET
MAGIC
PUT
THROW
```

Object:

knife

Description:

Sharp pointy thing.

Function:

Death and injury at close range.

Notes:

The knife works like a club: you go up to somebody and attack them with it. The difference is that the knife does more damage to avatars and less damage to property.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

`depends()`

Go:

`goTo()`

Stop:

`cease()`

Get:

`goToAndGet()`

Put:

`goToAndDropAt()`

Talk:

`broadcast()`

Reversed Do:

`strike()`

Initialization:

`nullInit()`

Destruction:

`smash()`

Asynchronous actions:

None.

Graphics:

Single static image of knife. Possibly multiple orientations (horizontal, vertical, maybe diagonal) to make the animation look right. Sound of knife striking avatar. Sound of knife striking object.

Host behavior:

`ATTACK`
`BASH`
`GET`
`PUT`

Object:

magic lamp

Description:

Just like Aladdin had...

Function:

Rub it and a genie appears to grant a wish.

Notes:

The magic lamp object represents both the magic lamp and the genie. You rub on the lamp and the genie appears. The first thing you say is interpreted as your wish and then the genie disappears. The genie operates like the oracle: it takes down your request; at some point one of our system people reviews the request and does something in response. It may take a while for your wish to be answered, and even then the answer may not be what you want or expect. The genie only grants one wish per person. Once the genie has noted your wish he disappears, and the lamp disappears with him. If you need to make another wish you must find another lamp (and another genie).

Styles:

None.

Properties:

```
lampState       /* What state of the wish granting process the lamp is in.
                   Possible states are MAGIC_LAMP_WAITING and
                   MAGIC_LAMP_GENIE. */
wisher          /* Who is making the wish (only the person who rubbed the lamp
                   may make the wish.) */
```

Class properties:

No class properties.

Host properties:

```
wishes          /* Queue of unprocessed wishes (what the wishes were and who
                   wished for them.) */
timeout         /* Delay process to kill genie when player waits too long. */
```

Command Behavior:

Do:

```
{
    /* If you are the one holding the lamp, rub it and make the genie
    appear.  Otherwise, depends. */
    if (holding(self)) {
        if (self.lampState == MAGIC_LAMP_WAITING) {
            @ self!RUB () → (success)
            if (success) {
                self.lampState = MAGIC_LAMP_GENIE
                self.wisher = avatar
                soundEffect(SOUND_POOF)
                balloonMessage(self, "Your wish is my command.")
            }
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:
```
goToAndDropAt()
```
Talk:
```
{
    /* Make a wish... */
    if (self.lampState == MAGIC_LAMP_GENIE && self.wisher == avatar) {
        @ self!WISH (text) → ()
        balloonMessage(self, "I'll see what I can do.")
        soundEffect(SOUND_WHOOSH)
        destroyObject(self)
    } else {
        broadcast()
    }
}
```
Reversed Do:
```
throw()
```
Initialization:
```
init(lampState, wisher)
{
    self.lampState = lampState
    self.wisher = wisher
}
```
Destruction:
```
smash()
```
Asynchronous actions:
```
RUB* ()
{
    self.lampState = MAGIC_LAMP_GENIE
    self.wisher = self.container
    soundEffect(SOUND_POOF)
}
WISH* ()
{
    soundEffect(SOUND_WHOOSH)
    destroyObject(self)
}
GIVEUP+ ()
{
    /* Asynchronous action when player has gone too long without wishing.
   (The genie won't wait forever!) */
    balloonMessage(self, "Time's up.  You lose.")
    soundEffect(SOUND_WHOOSH)
    destroyObject(self)
}
```
Graphics:

Image of lamp sitting there, and lamp with genie hovering over it. Animation of genie appearing and disappearing. Possible animation of genie hovering. Sound of genie appearing. Sound of genie vanishing.

Host behavior:
```
GET
PUT
RUB () → (success)
{
    if (holding(self) && self.lampState == MAGIC_LAMP_WAITING) {
```

```
        self.lampState = MAGIC_LAMP_GENIE
        self.wisher = avatar
        success = TRUE
        self.timeout = scheduleEvent(GenieGivesUp, GENIE_TIMEOUT_DELAY)
        # self → RUB* ()
    } else {
        success = FALSE
    }
}
THROW
WISH (text) → ()
{
    if (self.wisher == avatar) {
        cancelEvent(self.timeout)
        newWish = alloc(WISH_ENTRY)
        newWish->wish = text
        newWish->previousWish = self.wishes
        newWish->wisher = self.wisher
        self.wishes = newWish
        # self → WISH* ()
}
GenieGivesUp()
{
    # self → GIVEUP+ ()
    self → GIVEUP+ ()
    destroyObject(self)
}
```

Object:

magic staff

Description:

A stick about as tall as an avatar.

Function:

Generic magic talisman.

Notes:

Staves are usually magical. An staff will do something special and powerful. The magical function may vary. ***** The set of magical functions needs thought. ***** Staves are relatively rare and contain unusually powerful forms of magic. The general function of an staff will be to transform something or to perform some other specific action.

Styles:

```
STAFF_PLAIN, STAFF_JEWELED
```

Properties:

No properties.

Class properties:

No class properties.

Host properties:

```
magicType       /* What sort of magic this staff contains.  Used as case
                   switch for execution of magical behavior. */
```

Command Behavior:

Do:

```
doMagic()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
strike()
```

Initialization:

```
init(staffStyle)
{
    self.grstyle = staffStyle
}
```

Destruction:

```
smallExplosion()
```

Asynchronous actions:

None.

Graphics:

Single static image of staff for each staff style. Possible alternate orientations (horizontal, vertical, diagonal) to make the animation nice.

Host behavior:

```
ATTACK
BASH
GET
MAGIC
PUT
```

Object:

magic wand

Description:

Just like your fairy godmother has.

Function:

Generic magic talisman.

Notes:

Wands are usually magical. A wand will do something special but it not as powerful as a staff. The magical function may vary. ***** The set of magical functions needs thought. ***** Wands are relatively common as magical objects go (which is not very common at all) and usually contain fairly ordinary forms of magic. The general function of a wand will be to transform something or to perform some other specific action.

Styles:

```
WAND_PLAIN, WAND_STARRED
```

Properties:

No properties.

Class properties:

No class properties.

Host properties:

```
magicType       /* What sort of magic this wand contains.  Used as case
                   switch for execution of magical behavior. */
```

Command Behavior:

Do:

```
doMagic()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
strike()
```

Initialization:

```
init(wandStyle)
{
    self.grstyle = wandStyle
}
```

Destruction:

```
smallExplosion()
```

Asynchronous actions:

None.

Graphics:

Single static image of wand for each wand style. Possible alternate orientations (horizontal, vertical, diagonal) to make the animation nice.

Host behavior:

```
ATTACK
BASH
GET
MAGIC
PUT
```

Object:

mailbox

Description:

A conventional household roadside mailbox.

Function:

Interface to mail system.

Notes:

The mailbox lets us send and receive mail. A little red flag tells us that mail has arrived!

Styles:

None.

Properties:

```
mailArrived     /* Flag that there is mail waiting */
address         /* Mail address associated with this mailbox */
```

Class properties:

No class properties.

Host properties:

```
letters         /* Queue of unread letters. */
owner           /* The avatar whose mailbox this is. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
{
```

/ If there is mail for you, it is taken out of the mailbox you are popped into the sheet-of-paper handler to read it. Note that only a mailbox's owner can take mail out of it. */*

```
    if (goTo()) {
        if (self.mailArrived && emptyHanded(avatar) && address == avatar.address) {
            @ self!READMAIL () → (moreMail, text)
            self.mailArrived = moreMail
            message = createPaperObject(text)
            textEdit(message)
        }
    }
}
```

Put:

```
{
    sendMail(ADDRESS_CHECK)
}
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(mailArrived, address)
{
    self.mailArrived = mailArrived
    self.address = address
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
MAILARRIVED* ()
{
    /* Asynchronous message from host when new mail arrives */
    /* *** Trigger this. *** */
    self.mailArrived = TRUE
}
```

Graphics:

Two state image of mailbox, with little red flag up or down depending on whether or not there is mail waiting to be read.

Host behavior:

```
READMAIL () → (moreMail, text)
{
    if (self.mailArrived && emptyHanded(avatar) && owner == avatar) {
        text = letters->text
        deadLetter = letters
        letters = letters->nextLetter
        garbageCollect(deadLetter)
        mailArrived = (letters != NULL)
        moreMail = mailArrived
    } else {
        moreMail = FALSE
        text = ""
    }
}
SENDMAIL
```

Object:
matchbook
Description:
An empty book of matches
Function:
Has an ad that you can read.
Notes:
The matchbook is actually an information carrying device. There are never any matches.
Styles:
None.
Properties:
No properties.
Class properties:
No class properties.
Host properties:
`text`
Command Behavior:
Do:

```
{
    /* If holding the matchbook, read it.  Otherwise depends. */
    if (holding(self)) {
        @ self!README () → (text)
        balloonMessage(self, "%s", text)
    } else {
        depends()
    }
}
```

Go:
`goTo()`
Stop:
`cease()`
Get:
`goToAndGet()`
Put:
`goToAndDropAt()`
Talk:
`broadcast()`
Reversed Do:
`throw()`
Initialization:
`nullInit()`
Destruction:
`smash()`
Asynchronous actions:
None.
Graphics:
Single static image of matchbook.
Host behavior:

```
GET
PUT
README () → (text)
{
    if (holding(self)) {
        text = self.text
```

```
    } else {
        text = ""
    }
}
THROW
```

Object:
microphone
Description:
A reporter’s microphone.
Function:
Prop.
Notes:
This object doesn’t actually do anything (yet).
Styles:
None.
Properties:
No properties.
Class properties:
No class properties.
Host properties:
No properties.
Command Behavior:
Do:
`depends()`
Go:
`goTo()`
Stop:
`cease()`
Get:
`goToAndGet()`
Put:
`goToAndDropAt()`
Talk:
`broadcast()`
Reversed Do:
`throw()`
Initialization:
`nullInit()`
Destruction:
`smash()`
Asynchronous actions:
None.
Graphics:
Single static image of microphone.
Host behavior:
`GET`
`PUT`
`THROW`

Object:

motorcycle

Description:

Your basic motorcyle.

Function:

Carries one avatar around.

Notes:

The motorcycle is a vehicle that can travel on roads and sidewalks. It can only carry one avatar and it can't carry any other objects except what is already on the avatar's person.

Styles:

None.

Properties:

```
inhabitants[CYCLE_SIZE] /* The avatar riding in it.  This is NULL if there is
                    no rider. */
moving          /* A flag that the motorcycle is in motion. */
targetX         /* Motion destination X position. */
targetY         /* Motion destination Y position. */
```

Class properties:

```
maxOccupants = CYCLE_SIZE = 1
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
climbInOrOut()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(moving, targetX, targetY, inhabitant)
{
    self.moving = moving
    self.targetX = targetX
    self.targetY = targetY
    self.inhabitants[0] = inhabitant
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
DRIVE*
```

Graphics:

Three static images of motorcyle: side view, front view and back view. Sound of engine.

Host behavior:

None.

Object:

movie/television camera

Description:

The basic tool of journalism and drama.

Function:

Provides a way to record ongoing events.

Notes:

When the camera is on, the host records everything that happens in the region so that it can be played back later.

Styles:

None.

Properties:

```
on              /* Flag that the camera is on. TRUE if it is. */
```

Class properties:

No class properties.

Host properties:

```
film            /* The record made by this camera. */
```

Command Behavior:

Do:

```
{
    if (!toggleSwitch(!elsewhere(self))) {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(on)
{
    self.on = on
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
OFF*
ON*
```

Graphics:

Two-state image of camera with indicator light on and off. Also, humming sound when it is on.

Host behavior:

```
GET
OFF
ON
PUT
THROW
```

/* * Need some way to record passing events, and some way to play them back again. *** */**

Object:

pants

Description:

A pair of pants.

Function:

Decorative. Helps distinguish one avatar from another.

Notes:

Pants are an item of clothing whose primary purpose is to personalize avatars. They also have pockets. There are a few different styles of pants which are available in a variety of patterns.

Styles:

```
PANTS_LONG, PANTS_SHORT
```

Properties:

```
pattern         /* What pattern the pants cel should be painted with. */
contents        /* A list of the objects in the pants pockets. */
```

Class properties:

```
location = LEGS         /* Where this is worn. */
capacity = ?            /* How many things this can hold. */
displayContents = FALSE /* Whether to show what's in this. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
wearOrUnpocket()
```

Put:

```
remove()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(pantsStyle, pattern, contentsVector)
{
    self.grstyle = pantsStyle
    self.pattern = pattern
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

For each style, we need a single static image of pants in heap, plus avatar animation cels showing pants on legs in front, side and back views.

Host behavior:

```
GET
PUT
THROW
```

Object:

paper

Description:

A piece of paper.

Function:

Can be written upon and then retrieved.

Notes:

Any piece of paper whose text begins ‘‘to *name*’’ can be sent as a mail message.

Styles:

None.

Properties:

```
text            /* The text that is on the piece of paper. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    if (holding(self)) {
        textEdit(self)
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
{
    if (adjacent(self) && holdingClass(PENCIL_CLASS)) {
        writeTo(self, text)   /* *** shakey *** */
    } else {
        broadcast()
    }
}
```

Reversed Do:

```
throw()
```

Initialization:

```
init(text)
{
    self.text = text
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image of piece of paper. Text display of paper contents.

Host behavior:

GET
PUT
THROW

Object:

parking meter

Description:

Coin operated parking meter.

Function:

Appears to control where you park your car.

Notes:

The parking meter is a fairly inert scenic object. You can drop Tokens into it and cause it to change its state from ‘‘EXPIRED’’ to ‘‘OK’’, but this doesn’t actually *do* anything.

Styles:

None.

Properties:

```
meterState      /* What state the parking meter is in.  Possible values are
                   METER_EXPIRED and METER_TICKING. */
```

Class properties:

No class properties.

Host properties:

```
take            /* How many tokens this parking meter has raked in. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
{
```

/ If the avatar has the money, take the price of a parking and set the meter running. Otherwise, drop whatever is in hand next to the machine. */*

```
    if (payCoinOpOrDrop(PARKING_COST, SOUND_JINGLE,
            SOUND_CLICK_CLUNK, SOUND_JINK, goTo)) {
        self.meterState = METER_TICKING
    }
}
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can’t happen */
```

Initialization:

```
init(meterState)
{
    self.meterState = meterState
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
PAY* ()
{
    spend(PARKING_COST)
    soundEffect(SOUND_CLICK_CLUNK)
    self.meterState = METER_TICKING
```

```
}
EXPIRE+ ()
{
    /* Asynchronous event when meter expires. */
    self.meterState = METER_EXPIRED
}
```

Graphics:

Two-state image of parking meter: expired and ticking.

Host behavior:

```
PAY () → (success)
{
    if (spend(PARKING_COST)) {
        self.meterState = METER_TICKING
        scheduleEvent(MeterExpire, METER_TIME_DELAY)
        success = TRUE
        # self → PAY* ()
    } else {
        success = FALSE
    }
}
MeterExpire()
{
    # self → EXPIRE+ ()
    self → EXPIRE+ ()
    self.meterState = METER_EXPIRED
}
```

Object:

pencil

Description:

Your basic pencil.

Function:

Writes on paper.

Notes:

The pencil object is required in order to edit the contents of a piece of paper. If you do not have the pencil, you may only read. It's a magic pencil: it never needs sharpening.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

`depends()`

Go:

`goTo()`

Stop:

`cease()`

Get:

`goToAndGet()`

Put:

`goToAndDropAt()`

Talk:

`broadcast()`

Reversed Do:

`throw()`

Initialization:

`nullInit()`

Destruction:

`smash()`

Asynchronous actions:

None.

Graphics:

Single static image of pencil. We may need two orientations, horizontal and vertical, for the animation of the avatar handling it to look right.

Host behavior:

`GET`
`PUT`
`THROW`

Object:

phone booth

Description:

The traditional walk-in phone booth with real-live working pay phone (the sort which are getting rarer and rarer these days).

Function:

Interface to the telephone system for avatars away from home.

Notes:

The phone booth is probably the single most complex object in the system, in terms of its behavior. However, it works pretty much like an ordinary pay phone in the ‘‘real’’ world.

Styles:

None.

Properties:

```
phoneNumber     /* The 2-byte phone number associated with this phone */
state           /* The state that the phone is currently in.  The possible
                   states that the phone may be in at any given time are
                   PHONE_READY, PHONE_RINGING, PHONE_OFF_HOOK, PHONE_ACTIVE,
                   PHONE_TALKING, PHONE_LINE_RING and PHONE_LINE_BUSY */
```

Class properties:

No class properties.

Host properties:

```
talker          /* Avatar connected to this phone. */
caller          /* The phone at the other end of the line. */
take            /* Total income from this booth, in Tokens. */
beeper          /* Beeper associated with this phone.  Always NULL. */
answeringMachine /* Answering machine associated with this phone.  Always
                   NULL. */
owner           /* Avatar who owns this phone.  Always NULL. */
```

Command Behavior:

Do:

```
{
    /* If in booth, hang up phone if not already hung up.  Answer if
    ringing.  Otherwise, depends. */
    hangUpOrAnswer(here(self))
}
```

Go:

```
enterOrExit()
```

Stop:

```
cease()
```

Get:

```
{
    /* Answer the phone if it's ringing, otherwise just pick up the
    receiver. */
    answerOrUnhook(here(self), PHONE_OFF_HOOK)
}
```

Put:

```
{
    /* If avatar has a token in hand, activate the phone.  If holding the
    receiver, hang up.  Otherwise drop whatever is in hand next to the
    phone booth. */
    if (self.state == PHONE_READY || (here(self) && self.state ==
            PHONE_OFF_HOOK)) {
        if (payCoinOpOrDrop(PHONE_CALL_COST, SOUND_JINGLE,
                SOUND_CLICK_CLUNK, SOUND_JINK, goEnter)) {
```

```
            self.state = PHONE_ACTIVE
            soundEffect(SOUND_DIAL_TONE)
        }
    } else if (here(self) && (self.state == PHONE_LINE_BUSY || self.state ==
            PHONE_LING_RING || self.state == PHONE_TALKING)) {
        hangUp()
        soundEffect(SOUND_JANGLE)
        refund(PHONE_CALL_COST, avatar)
    }
}
```

Talk:

```
{
```

/ If the phone is active (from dropping a token in it), interpret text as a phone call: first text message is phone number, further messages are conversation with the person at the other end. When you enter the number, the phone dials. If there is an answer, you can talk. If not, you get your tokens back (into avatar's hand) and the phone is deactivated when you hang up. If phone is not active, broadcast. */*

```
    dialOrTalk(here(self))
}
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(phoneNumber, state)
{
    self.phoneNumber = phoneNumber
    self.state = state
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
ANSWERED*
ANSWER*
HANG*
HUNGUP*
UNHOOK*
RING*
DIAL*
SPEAK*
```

Graphics:

Single static image of phone booth. Cel for receiver so we can show it in the avatar's hand. Sound of phone in booth ringing. Sound of phone ringing at other end of phone line. Busy signal sound. Dial tone. Dialing sounds (touch tones or rotary clanks). Clicking and clunking when the receiver is picked up or hung up or when the person at the other end hangs up. Sounds of coins being deposited, refunded and swallowed.

Host behavior:

```
ANSWER
DIAL
HANG
PAY () → (success)
{
    if (spend(PHONE_CALL_COST) && self.state == PHONE_OFF_HOOK) {
        self.state = PHONE_ACTIVE
        success = TRUE
```

```
    } else {
        success = FALSE
    }
}
TALK
UNHOOK
```

Object:

picture

Description:

A picture to display.

Function:

Displays artwork or text on wall or in space.

Notes:

A picture is represented by a set of imagery that is part of its state information. Due to the large overhead associated with such objects, pictures will have to be either small or infrequent.

Styles:

None.

Properties:

```
size            /* How big the picture is.  This is a binary value: TRUE if
                   the picture is too large for an avatar to pick up. */
picture         /* The contents of the picture, represented as a conventional
                   MicroCosm graphic (except that the data comes from the host
                   rather than from the internal object definitions. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
{
    /* Grab it if you can (if it's not too big). */
    if (goTo()) {
        if (!size) {
            get()
        }
    }
}
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(size, picture)
{
    self.size = size
    self.picture = picture
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

A single static image which is rendered out of the `picture` state variable.

Host behavior:

```
GET     /* *** Need special check to handle size. *** */
PUT
THROW
```

Object:

plant

Description:

Your basic generic plant.

Function:

Scenic element. Obstruction.

Notes:

This is a fairly inert scenic element, provided almost entirely for visual appeal. We may wish to consider giving certain styles of plants magical powers (healing, etc.) in addition, but such uses are not addressed here.

Styles:

```
PLANT_WEED, PLANT_GRASS, PLANT_FLOWER, PLANT_REED, PLANT_VINE, PLANT_GRAIN
```

Properties:

```
uprootable      /* Flag whether or not plant can be uprooted by hand */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
{
    /* Get the plant, but only if it is uprootable */
    if (self.uprootable) {
        goToAndGet()
    } else {
        goTo()
    }
}
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(plantStyle, uprootable)
{
    self.grstyle = plantStyle
    self.uprootable = uprootable
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image for each style of plant.

Host behavior:

```
GET    /* *** Need special check for uprootability *** */
```

```
PUT
THROW
```

Object:

pond

Description:

A small (within the region) body of water.

Function:

Water source. Water obstruction.

Notes:

***** We need to figure out how we are going to indicate where the pond is and where it is not. *****

Styles:

None.

Properties:

```
coverage        /* Somehow indicate what areas the pond covers */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndFill()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(coverage)
{
    self.coverage = coverage
}
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

None.

Graphics:

Pond image on ground, rendered during background processing.

Host behavior:

None.

Object:

radio

Description:

Your basic boom box.

Function:

Lets us hear the local radio station.

Notes:

The radio only receives one channel, *Radio Free MicroCosm*. When the radio is turned on, it plays whatever is on the radio at the time. When it is off it does nothing.

Styles:

None.

Properties:

```
on              /* Flag that the radio is on.  TRUE if it is. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    if (!toggleSwitch(!elsewhere(self))) {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(on)
{
    self.on = on
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
OFF*
ON*
PLAY+ (score)
{
    /* Asynchronous function when something comes in to be played. */
    /* *** Trigger this. *** */
    playMusic(score)
}
ANNOUNCE+ (text)
{
```

```
        /* Asynchronous function when an announcement comes over the radio. */
        /* *** Trigger this. *** */
          balloonMessage(self, "%s", text)
          playSpeech(text)
    }
```

Graphics:

Host behavior:

```
    GET
    OFFPLAYER
    ONPLAYER
    PUT
    THROW
```

Object:

region

Description:

A place in the world.

Function:

The basic building block of **MicroCosm** topography.

Notes:

The region is not really an object, but we have an object that represents it so that the host can send messages to it. Basically, all it does is contain things. It also has a few other properties that are used for background graphics rendering. Note that some of the region's generic properties are interpreted slightly differently. In particular, the `y` position encodes the vertical screen position of the horizon line.

Styles:

```
REGION_PLAIN, REGION_URBAN, REGION_FOREST, REGION_DESERT, REGION_INTERIOR
```

Properties:

```
contents        /* A list of the objects in the region. */
width           /* Width of the region, in feet. */
depth           /* Depth of the region, in feet. */
classGroup      /* Region-specific object class partition associate with this
                   region (i.e., which set of 128 region-specific object
                   classes do we use here?). */
avatar          /* Which avatar in the region represents the player. */
```

Class properties:

No class properties.

Host properties:

```
exclusion       /* Flag that entry to this region is blocked by security
                   device. */
regionNumber    /* The 2-byte identifier associated with this region. */
connectivity[4] /* The other regions West, East, North and South of here. */
```

Command Behavior:

None: it is not a selectable object on the screen.

Initialization:

```
init(terrainStyle, horizon, width, depth, regionNumber, classGroup, contentsVector)
{
    self.grstyle = terrainStyle
    self.horizon = horizon
    self.width = width
    self.depth = depth
    self.regionNumber = regionNumber
    self.classGroup = classGroup
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

None.

Graphics:

None. Everything falls out of the objects that are here.

Host behavior:

```
DESCRIBE () → (regionDescriptor)
{
    regionDescriptor = vectorize(self);
}
```

Object:

ring

Description:

Your basic magic ring.

Function:

Generic magic talisman.

Notes:

Rings are always magical. A ring will do something special and powerful. The magical function may vary. ***** The set of magical functions needs thought. ***** Rings are extremely rare and contain unusually powerful forms of magic (not always good). The general function of a ring will be to confer some particular power or ability on the avatar wearing it.

Styles:

None.

Properties:

No properties.

Class properties:

```
location = HAND /* Where this is worn. */
```

Host properties:

```
magicType       /* What sort of magic this ring contains.  Used as case
                   switch for execution of magical behavior. */
```

Command Behavior:

Do:

```
doMagic()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
wear()
```

Put:

```
remove()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
nullInit()
```

Destruction:

```
smallExplosion()
```

Asynchronous actions:

None.

Graphics:

Single static image of ring. Cels for ring on avatar's finger in front, back and side views.

Host behavior:

```
GET
MAGIC
PUT
THROW
```

Object:

river

Description:

Body of water flowing through a region.

Function:

Water source. Linear water obstruction.

Notes:

This poses essentially the same problem that the pond does: how do we indicate where the water is and where it is not? We may, in fact, want to merge the mechanisms of the two so that there is just a single type of body-of-water object. However, for rivers we may want to have currents so things will drift (***** unsolved problem *****).

Styles:

None.

Properties:

```
path            /* Somehow indicate where the river flows */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndFill()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(path)
{
    self.path = path
}
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

None.

Graphics:

River image on ground, rendered during background processing.

Host behavior:

None.

Object:
rock
Description:
Your basic rock.
Function:
Scenic element. If small can be picked up, thrown.
Notes:
Rocks are mostly for visual variation, but can also be used as weapons.
Styles:
`ROCK_SMALL, ROCK_MEDIUM, ROCK_LARGE, ROCK_HUGE`
Properties:
No properties.
Class properties:
No class properties.
Host properties:
No properties.
Command Behavior:
Do:
`depends()`
Go:
`goTo()`
Stop:
`cease()`
Get:

```
{
    /* Get the rock if it's not too heavy. */
    if (self.grstyle == ROCK_SMALL || self.grstyle == ROCK_MEDIUM) {
        goToAndGet()
    } else {
        goTo()
    }
}
```

Put:
`goToAndDropAt()`
Talk:
`broadcast()`
Reversed Do:
`throw()`
Initialization:

```
init(weight)
{
    self.weight = weight
}
```

Destruction:
`smash()`
Asynchronous actions:
None.
Graphics:
One static image of rock for each weight. Sound of rock hitting something.
Host behavior:
`GET` **/* *** Need special check to handle weights. *** */**
`PUT`
`THROW`

Object:

roof

Description:

The roof of a building.

Function:

Graphic element in buildings.

Notes:

A roof is a single trapezoid that sits on top of a building. Roof objects are not found in isolation but only as components of building objects. It is a fairly inert background object. Once painted on the background it is of little consequence except visually. Unlike ground and sky, which are pseudo-objects, roof objects really exist, but they are not known to the host, since they never take part in any host interactions. Rather, they are created internally by building objects (which *are* known to the host).

Styles:

```
ROOF_FLAT, ROOF_GABLED, ROOF_A_FRAME
```

Properties:

```
size            /* How big the roof is.  This is basically a length.  The
                   height is determined algorithmically from this value and
                   from the style of roof that it is. */
base            /* How high above the ground the bottom of the roof starts. */
roofPattern     /* The color and texture that the roof appears. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
noEffect()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
noEffect()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(roofStyle, size, base, roofPattern)
{
    self.grstyle = roofStyle
    self.size = size
    self.base = base
    self.roofPattern = roofPattern
}
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

None.

Graphics:

Roof image as part of building, rendered by background processor when it draws the whole building.

It is drawn as a texture mapped trapezoid (ROOF_FLAT comes out as a rectangle, ROOF_GABLED comes out as a trapezoid and ROOF_A_FRAME comes out as a triangle). Note that roofs are always viewed face-on, never edge-wise.

Host behavior:

None.

Object:

rubber ducky

Description:

Your basic bathtub toy.

Function:

To amuse people in the hot tub.

Notes:

This object doesn’t actually do anything (yet).

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

`depends()`

Go:

`goTo()`

Stop:

`cease()`

Get:

`goToAndGet()`

Put:

`goToAndDropAt()`

Talk:

`broadcast()`

Reversed Do:

`throw()`

Initialization:

`nullInit()`

Destruction:

`smash()`

Asynchronous actions:

None.

Graphics:

Single static image of rubber ducky.

Host behavior:

`GET`
`PUT`
`THROW`

Object:

safe

Description:

Your basic office or household safe.

Function:

Stores things under lock and key.

Notes:

The safe is both a piece of furniture, for scenic purposes, and a container. It has two states, open and closed, and it can be locked. When it is open, anyone in the region can see the contents, thus a safe may only be opened if there are enough free object identifiers available to identify the complete contents to all the players. The safe operates just like a chest of drawers with the exception that it can be locked.

Styles:

None.

Properties:

```
contents        /* Contents list: a list of the objects that the safe
                   contains.  This will always be NULL if the safe is
                   closed. */
open            /* Flag telling whether the safe is open.  TRUE if open. */
key             /* Two-byte number indicating the key required for this
                   safe. */
unlocked        /* Flag that lock is unlocked.  TRUE if it is. */
```

Class properties:

```
capacity = ?            /* How many things this can hold. */
displayContents = FALSE /* Whether to show what's in it. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
adjacentOpenCloseContainer()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndPickFromIfOpen()
```

Put:

```
goToAndDropIntoIfOpen()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(open, key, unlocked, contentsVector)
{
    self.open = open
    self.key = key
    self.unlocked = unlocked
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

Two-state image of safe: safe open and safe closed.

Host behavior:

```
CLOSECONTAINER
OPENCONTAINER
```

Object:

security device

Description:

Another miscellaneous gadget.

Function:

Provides a way to make a region safe from intrusion.

Notes:

The security device provides a way to protect your avatar and property from harm. When the security device is turned on, no one else can enter or leave the region it is in. It only works in selected regions (like hotel rooms). The device indicates its operation by means of both an indicator light and a humming sound.

Styles:

None.

Properties:

```
on              /* Flag that device is turned on. TRUE if it is. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
{
    if (!toggleSwitch(!elsewhere(self))) {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(on)
{
    self.on = on
}
```

Destruction:

```
mediumExplosion()
```

Asynchronous actions:

```
OFF*
ON*
```

Graphics:

Two-state image of device with indicator light on and off. Also, humming sound when it is on.

Host behavior:

```
GET
OFF () → ()
{
```

```
    if (!elsewhere(self) && self.on) {
        self.on = FALSE
        region.exclusion--
        # self → OFF* ()
    }
}
ON () → ()
{
    if (!elsewhere(self) && !self.on) {
        self.on = TRUE
        region.exclusion++
        # self → ON* ()
    }
}
PUT
THROW
```

Object:

sensor

Description:

Another miscellaneous gadget.

Function:

Tells some property of a region, object or avatar.

Notes:

The sensor can scan for information which is not always directly perceptible to an avatar or, for that matter, to the player's home system. What it scans for depends on what type of sensor it is. For example, a weapon sensor will tell if any of the avatars in the region is carrying a weapon, even if the weapon is inside a backpack or similar container. The sensing is performed by the host, so it is guaranteed to be ''honest''. ***** The set of things that sensors sense needs further thought. *****

Styles:

`SENSOR_WEAPON, SENSOR_LIFE_REMOTE, SENSOR_MAGIC, SENSOR_MAGIC_REMOTE, etc.`

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
{
    /* Do a scan with the detector and make a sound appropriate to what was
    found. */
    if (!elsewhere(self)) {
        @ self!SCAN () → (detection)
        if (detection) {
            soundEffect(SOUND_DETECT)
        } else {
            soundEffect(SOUND_NO_DETECT)
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(sensorStyle)
{
    self.grstyle = sensorStyle
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
SCAN* (detection)
{
    if (detection) {
        soundEffect(SOUND_DETECT)
    } else {
        soundEffect(SOUND_NO_DETECT)
    }
}
```

Graphics:

Single static image of sensor. Sounds of detection and non-detection.

Host behavior:

```
GET
PUT
SCAN () → (detection)
{
    if (holding(self)) {
        detection = (*sensorActions[self.style])()
        # self → SCAN* (detection)
    } else {
        detection = FALSE
    }
}
THROW
```

Object:

shirt

Description:

Your basic shirt.

Function:

Decorative. Helps distinguish one avatar from another.

Notes:

A shirt is a clothing object whose only purpose is to personalize avatars. A shirt operates like a jacket, except that it has no pockets. Unlike the real world, it is not possible to wear a shirt and a jacket at the same time. There are a couple of different styles of shirts.

Styles:

```
SHIRT_T, SHIRT_LONG_SLEEVE, SHIRT_SHORT_SLEEVE
```

Properties:

```
pattern         /* The pattern the shirt cels should be painted with. */
```

Class properties:

```
location = TORSO /* Where this is worn. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
wear()
```

Put:

```
remove()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(shirtStyle, pattern)
{
    self.grstyle = shirtStyle
    self.pattern = pattern
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

For each style of shirt, a single static image of shirt in heap, plus avatar animation cels showing shirt on torso in front, side and back views.

Host behavior:

```
GET
PUT
THROW
```

Object:

shoes

Description:

A pair of shoes.

Function:

Decorative. Helps distinguish one avatar from another.

Notes:

A pair of shoes is a clothing object whose only purpose is to personalize avatars. It is represented by a couple of cels that simply follow the avatar's feet.

Styles:

```
SHOES_RUNNING, SHOES_DRESS, SHOES_BOOTS
```

Properties:

No properties.

Class properties:

```
location = FEET          /* Where this is worn. */
```

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
wear()
```

Put:

```
remove()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(shoeStyle)
{
    self.grstyle = shoeStyle
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Cels of shoes for each style of shoes in side, front and back views.

Host behavior:

```
GET
PUT
THROW
```

Object:

sidewalk

Description:

Your basic suburban sidewalk.

Function:

Scenic element. Can be walked on (keeps feet off the lawn!).

Notes:

Each sidewalk object is a single trapezoid of sidewalk material. The sides are always parallel to the edges of the region. It is a fairly inert background object. Once painted on the ground, it is of no consequence except visually, and behaves exactly like ground. Whether we in fact wish to have it disappear once painted on the ground and have the ground behavior come into play when sidewalk is pointed at or whether we want to simply have it execute exactly the same behavior protocol as ground is unclear.

Styles:

None.

Properties:

```
width           /* Section width from the corner */
height          /* Section length from the corner */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goToCursor()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToCursorAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(width, height)
{
    self.width = width
    self.height = height
}
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

None.

Graphics:

Sidewalk image on ground, rendered during background processing, as a single, texture-mapped trapezoid.

Host behavior:

None.

Object:

sign

Description:

A standard sign.

Function:

For public safety and information.

Notes:

The sign is a fairly inert scenic object, provided almost entirely for visual appeal.

Styles:

`SIGN_EXIT, SIGN_BILLBOARD, SIGN_FOR_SALE`

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
nullInit()
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image of sign (front view).

Host behavior:

None.

Object:

skateboard

Description:

Your basic skateboard.

Function:

Carries one avatar in a silly manner.

Notes:

The skateboard is a vehicle that can travel on roads and sidewalks. It can only carry one avatar and it can't carry any other objects except what is already on the avatar's person.

Styles:

None.

Properties:

```
inhabitants[SKATE_SIZE] /* The avatar riding in it.  This is NULL if there is
                    no rider. */
moving          /* A flag that the skateboard is in motion. */
targetX         /* Motion destination X position. */
targetY         /* Motion destination Y position. */
```

Class properties:

```
maxOccupants = SKATE_SIZE = 1
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
climbInOrOut()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(moving, targetX, targetY, inhabitant)
{
    self.moving = moving
    self.targetX = targetX
    self.targetY = targetY
    self.inhabitants[0] = inhabitant
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
WALK*
```

Graphics:

Three static images of skateboard: side view, front view and back view.

Host behavior:

```
GET
PUT
THROW
```

Object:

skirt

Description:

Your basic skirt.

Function:

Decorative. Helps distinguish one avatar from another.

Notes:

Skirt are items of clothing whose primary purpose is to personalize avatars. They also have pockets, which are useful for holding things without resorting to hands. There is only one sort of skirt, but it comes in a variety of patterns. Skirts function exactly like pants, except that they are perhaps more appropriate for a female avatar.

Styles:

None.

Properties:

```
pattern         /* The pattern the skirt cels should be painted with. */
contents        /* A list of the objects in the skirt's pockets.  NULL if the
                   pockets are empty. */
```

Class properties:

```
location = LEGS         /* Where this is worn. */
capacity = ?            /* How many things this can hold. */
displayContents = FALSE /* Whether to show what's in it. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
wearOrUnpocket()
```

Put:

```
remove()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(pattern, contentsVector)
{
    self.pattern = pattern
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

Single static image of skirt in heap, plus avatar animation cels showing skirt on legs in front, side and back views.

Host behavior:

GET

PUT

THROW

Object:

sky

Description:

The basic background above the horizon.

Function:

Can be pointed at, returning a location.

Notes:

This is not really an object, but is treated as one to make the player interface clean.

Styles:

None.

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
noEffect()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
noEffect()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
nullInit()
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

None.

Graphics:

Blue (or other sky color) from top of graphics window to the scenic horizon. Scenic background color from scenic horizon down to the true horizon. Scenic background horizon line is a jaggy line procedurally determined from the terrain style. Drawn by the background rendering initialization routines.

Host behavior:

None.

Object:

stereo

Description:

Your basic tape deck.

Function:

Plays music of your choice.

Notes:

The stereo is like the radio, but instead of playing whatever comes over the ‘‘air’’ it plays tapes. Tapes contain specific pieces of music which the host knows about.

Styles:

None.

Properties:

```
on              /* Flag that stereo is playing.  TRUE if it is. */
tape            /* What tape is in the stereo.  NULL if none is. */
```

Class properties:

```
capacity = 1    /* Can only hold 1 tape. */
displayContents = FALSE /* Whether to show tape. */
```

Host properties:

```
playPointer     /* Where we are in the current tape. */
```

Command Behavior:

Do:

```
{
    if (!toggleSwitch(holding(self))) {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
{
    /* If holding the stereo, pop the tape out.  Otherwise, get it. */
    if (holding(self) && !self.on) {
        if (self.tape != NULL) {
            changeContainers(self.tape, region, avatar.x, avatar.y)
            self.tape = NULL
            @ self!UNLOAD () → ()
        }
    } else {
        goToAndGet()
    }
}
```

Put:

```
{
    /* If holding a tape and the stereo has none, put the tape in the
    stereo.  Otherwise, just drop whatever next to the stereo. */
    if (holdingClass(TAPE_CLASS) && self.tape == NULL && !self.on && goTo()) {
        @ self!LOAD (avatar.inHand.noid) → (success)
        if (success) {
            avatar.action = PUT
            subjectObject = avatar.inHand
            changeContainers(avatar.inHand, self, NULL, TAPE)
        }
```

```
        } else {
            goToAndDropAt()
        }
    }
```

Talk:

```
    broadcast()
```

Reversed Do:

```
    throw()
```

Initialization:

```
    init(on, tape)
    {
        self.on = on
        self.tape = tape
    }
```

Destruction:

```
    smash()
```

Asynchronous actions:

```
    PLAY+ (score)
    {
        /* Asynchronous function when something comes in to be played. */
        /* *** Trigger this. *** */
        if (self.on) {
            playMusic(score)
        }
    }
```

Graphics:

Single static image of tape deck.

Host behavior:

```
    GET
    LOAD (tapeId) → (success)
    {
        tape == ↑tapeId
        if (holding(tape) && tape.class == TAPE_CLASS && adjacent(self)
                && !self.on && self.tape == NULL) {
            self.tape = tape
            changeContainers(tape, self, NULL, TAPE)
            success = TRUE
        } else {
            success = FALSE
        }
    }
    OFFPLAYER
    ONPLAYER
    PUT
    THROW
    UNLOAD () → ()
    {
        if (holding(self) && !self.on && self.tape != NULL) {
            changeContainers(self.tape, region, avatar.x, avatar.y)
            self.tape = NULL
        }
    }
```

Object:

street

Description:

Your basic roadway.

Function:

Can be walked on. Carries ground vehicles.

Notes:

Each street object is a single trapezoid of street material. The sides are always parallel to the edges of the region. It is relatively inert, like sidewalk, but does have some special properties. It behaves just like ground. However, cars can only drive on the surfaces of streets (unless they are special, off-road vehicles), and so in that sense it is special.

Styles:

None.

Properties:

```
width           /* Section width from the corner */
height          /* Section length from the corner */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goToCursor()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToCursorAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(width, height)
{
    self.width = width
    self.height = height
}
```

Destruction:

```
nullDestroy()
```

Asynchronous actions:

Cars may drive on streets. (They may not drive on other sorts of surfaces.)

Graphics:

Road image on ground, rendered during background processing, as a single, texture-mapped trapezoid.

Host behavior:

None.

Object:

streetlamp

Description:

Conventional streetlamp.

Function:

Scenic element. Provides light at night.

Notes:

The **MicroCosm** follows a cycle of day and night. When it is night-time, the screen is displayed in darkened colors, which makes things hard to see. Having a streetlamp in the region makes the region appear like daytime, even if it is night elsewhere.

Styles:

`STREETLAMP_GASLIGHT, STREETLAMP_MODERN`

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(lampStyle)
{
    self.grstyle = lampStyle
}
```

Destruction:

```
{
    smash()
    lightLevel--
}
```

Asynchronous actions:

None.

Graphics:

A single static image for each style of streetlamp.

Host behavior:

None.

Object:

table

Description:

A common table.

Function:

Can support things (hold them off the floor).

Notes:

The table is a form of container whose contents are always visible and accessible (i.e., it is permanently ‘‘open’’). Things can be placed on it and picked up off of it.

Styles:

`TABLE_KITCHEN, TABLE_COFFEE, TABLE_DINING`

Properties:

```
contents        /* Contents list: a list of the objects on the table.  */
```

Class properties:

```
capacity = ?            /* How many things this can hold. */
displayContents = TRUE  /* Whether to show what's in this. */
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndPickFrom()
```

Put:

```
goToAndDropOnto()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(tableStyle, contentsVector)
{
    self.grstyle = tableStyle
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smashWithContents()
```

Asynchronous actions:

None.

Graphics:

A single static image of a table for each table style.

Host behavior:

None.

Object:
tape

Description:
A cassette tape.

Function:
Can be played in the stereo tape deck.

Notes:
The tape simply serves to identify which piece of music the stereo should play.

Styles:
None.

Properties:
```
music           /* 2-byte identifier telling what music is on this tape. */
```

Class properties:
No class properties.

Host properties:
No other properties.

Command Behavior:

Do:
```
{
    /* If holding the tape, read the label.  Otherwise depends. */
    if (holding(self)) {
        @ self!READLABEL () → (text)
        balloonMessage(self, "%s", text)
    } else {
        depends()
    }
}
```

Go:
```
goTo()
```

Stop:
```
cease()
```

Get:
```
goToAndGet()
```

Put:
```
goToAndDropAt()
```

Talk:
```
broadcast()
```

Reversed Do:
```
throw()
```

Initialization:
```
init(music)
{
    self.music = music
}
```

Destruction:
```
smash()
```

Asynchronous actions:
None.

Graphics:
Single static image of tape cassette.

Host behavior:
```
GET
PUT
READLABEL () → (text)
```

```
{
    if (holding(self)) {
        text = music[self.music]->label
    } else {
        text = ""
    }
}
THROW
```

Object:
teddy bear
Description:
A soft, cuddly teddy bear.
Function:
A great security enhancer.
Notes:
This is the most valuable item in the **MicroCosm**, as there is only one of them.
Styles:
None.
Properties:
No properties.
Class properties:
No class properties.
Host properties:
No properties.
Command Behavior:
Do:
`depends()`
Go:
`goTo()`
Stop:
`cease()`
Get:
`goToAndGet()`
Put:
`goToAndDropAt()`
Talk:
`broadcast()`
Reversed Do:
`throw()`
Initialization:
`nullInit()`
Destruction:
`mediumExplosion()`
Asynchronous actions:
None.
Graphics:
Single static image of teddy bear.
Host behavior:
`GET`
`PUT`
`THROW`

Object:

telephone

Description:

The household telephone.

Function:

Remote communications with other players on-line.

Notes:

The household telephone operates similarly to the phone booth pay-phone, but it is not so complicated because it doesn't have to collect money from the player in order to work.

Styles:

None.

Properties:

```
phoneNumber     /* The 2-byte phone number associated with this phone */
state           /* The state that the phone is currently in.  The possible
                   states that the phone may be in at any given time are
                   PHONE_READY, PHONE_RINGING, PHONE_ACTIVE, PHONE_TALKING,
                   PHONE_LINE_RING and PHONE_LINE_BUSY */
```

Class properties:

No class properties.

Host properties:

```
talker          /* The avatar currently connected to this phone. */
caller          /* The phone at the other end of the line. */
beeper          /* Beeper associated with this phone. */
answeringMachine /* Answering machine associated with this phone. */
owner           /* Avatar associated with this phone. */
```

Command Behavior:

Do:

```
{
    /* If at phone, hang it up if not already hung up.  Answer if ringing.
   Otherwise, depends. */
    hangUpOrAnswer(!elsewhere(self))
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
{
    /* Answer the phone if it's ringing, otherwise just pick up the
   receiver. */
    answerOrUnhook(goTo(), PHONE_ACTIVE)
}
```

Put:

```
{
    /* If holding the receiver, hang up.  Otherwise drop whatever is in
   hand next to the phone. */
    if (!elsewhere(self) && (self.state == PHONE_ACTIVE || self.state ==
            PHONE_LINE_BUSY || self.state == PHONE_LING_RING || self.state ==
            PHONE_TALKING)) {
        hangUp()
    } else {
        goToAndDropAt()
    }
}
```

Talk:

```
{
    /* If the phone is active, interpret text as a phone call: first text
    message is phone number, further messages are conversation with the
    person at the other end.  When you enter the number, the phone dials.
    If there is an answer, you can talk.  If not, the phone is deactivated
    when you hang up.  If phone is not active, broadcast. */
     dialOrTalk(!elsewhere(self))
}
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(phoneNumber, state)
{
    self.phoneNumber = phoneNumber
    self.state = state
}
```

Destruction:

```
smash()
```

Asynchronous actions:

```
ANSWERED*
ANSWER*
HANG*
HUNGUP*
UNHOOK*
RING*
DIAL*
SPEAK*
```

Graphics:

Single static image of phone. Cel for receiver so we can show it in the avatar's hand. Sound of phone ringing. Sound of phone ringing at other end of phone line. Busy signal sound. Dial tone. Dialing sounds (touch tones or rotary clanks). Clicking and clunking when the receiver is picked up or hung up or when the person at the other end hangs up.

Host behavior:

```
ANSWER
DIAL
HANG
TALK
UNHOOK
```

Object:

teleport booth

Description:

Like a phone booth, but it carries all of you instead of just your voice.

Function:

Zaps avatars and their possessions elsewhere instantaneously.

Notes:

The teleport booth is fairly complex. However, it operates very much like a pay phone. The only difference is that you get transported to the destination. This means that you don't have to wait for somebody to answer or suffer a busy signal: you just go. It is also more expensive than a phone call.

Styles:

None.

Properties:

```
boothNumber     /* The 2-byte port number associated with this port. */
state           /* The state that the port is currently in.  The possible
                   states are PORT_READY and PORT_ACTIVE. */
```

Class properties:

No class properties.

Host properties:

```
take            /* Total income from this booth, in Tokens. */
```

Command Behavior:

Do:

```
depends()
```

Go:

```
enterOrExit()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
{
    /* If avatar has a token in hand and the port is not already active,
    activate the port.  Otherwise drop whatever is in hand next to the
    booth. */
    if (self.state == PORT_READY) {
        if (payCoinOpOrDrop(TELEPORT_COST, SOUND_JINGLE,
                SOUND_CLUNK_BUZZ, SOUND_JINK, goEnter)) {
            self.state = PORT_ACTIVE
            soundEffect(SOUND_DIAL_TONE)
        }
    }
}
```

Talk:

```
{
    /* If the teleporter is active (from dropping a token in it),
    interpret text as a teleport booth number, and teleport the avatar and
    everything he is carrying to the teleport booth dialed.  If the booth
    is not active, broadcast. */
    if (here(self) && self.state == PORT_ACTIVE) {
        @ self!ZAPTO (text) → (success)
        soundEffect(SOUND_DIAL, text)
        if (success) {
            self.state = PORT_READY
            soundEffect(SOUND_ZAP_OUT)
```

```
            }
        } else {
            broadcast()
        }
    }
```

Reversed Do:

```
    /* This can't happen */
```

Initialization:

```
    init(boothNumber, state)
    {
        self.boothNumber = boothNumber
        self.state = state
    }
```

Destruction:

```
    mediumExplosion()
```

Asynchronous actions:

```
    PAY* ()
    {
        spend(TELEPORT_COST)
        soundEffect(SOUND_CLUNK_BUZZ)
        self.state = PORT_ACTIVE
        soundEffect(SOUND_DIAL_TONE)
    }
    ZAPTO* (avatar)
    {
        self.state = PORT_READY
        soundEffect(SOUND_ZAP_OUT)
    }
    ZAPIN* ()
    {
        /* Asynchronous function when somebody arrives from someplace else. */
        /* *** Trigger this. *** */
        soundEffect(SOUND_ZAP_IN)
    }
```

Graphics:

Single static image of teleport booth. Sound of dialing. Sound of yourself zapping out to someplace else. Sound of somebody else zapping in from elsewhere.

Host behavior:

```
    PAY () → (success)
    {
        if (self.state == PORT_READY && spend(TELEPORT_COST)) {
            self.state = PORT_ACTIVE
            success = TRUE
            # self → PAY* ()
        } else {
            success = FALSE
        }
    }
    ZAPTO (text) → (success)
    {
        if (here(self) && self.state == PORT_ACTIVE) {
            destination = lookupTeleportAddress(text)
            if (destination != NULL) {
                goToNewRegion(avatar, destination)
```

```
            success = TRUE
            self.state = PORT_READY
            # self → ZAPTO* (avatar.noid)
        } else {
            success = FALSE
        }
    } else {
        success = FALSE
    }
}
```

Object:

ticket

Description:

A ticket to the show.

Function:

Your avatar's admission to special events.

Notes:

Tickets are used to control admission to certain regions. You have to have the ticket that corresponds to an event in such a regions in order to be let in.

Styles:

None.

Properties:

```
event           /* What event this ticket is admission for. */
```

Class properties:

No class properties.

Host properties:

No additional properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
init(event)
{
    self.event = event
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image of ticket.

Host behavior:

```
GET
PUT
THROW
```

Object:

tokens

Description:

Filthy lucre.

Function:

Money in the MicroCosm.

Notes:

The standard unit of currency in the **MicroCosm** is the Token. On one side it says*Good F or One Fare*. On the other side it says*F iat Lucre*. A single token object can denote any amount of money. The `denomination` property tells how much money a token is currently representing. Certain machines in the **MicroCosm** world are coin operated. Dropping a token into them will cause the appropriate amount of money to be consumed. If there is any left over it is left in the avatar's hand. Other sorts of transactions require that the quantity be specified by the player. You can open up a token almost as if it was a container object and interactively select the amount of money you wish to extract. A second token object is created with this denomination and placed wherever indicated, while the first token object has its denomination reduced accordingly and is left in the player's hand.

Styles:

None.

Properties:

```
denomination    /* How much money this token represents. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
{
    /* Select an amount of money and fork it over. */
    amount = selectDenomination(self.denomination)
    @ self!PAY (target, amount) → (success, actualTarget, x, y)
    if (success) {
        self.denomination -= amount
        if (self.denomination == 0) {
            destroyObject(self)
        }
        if (actualTarget.class == TOKEN_CLASS) {
            actualTarget.denomination += amount
        } else {
            newToken = createObject(TOKEN_CLASS, actualTarget, x, y)
            newToken.denomination = amount
        }
        soundEffect(SOUND_JINGLE)
```

```
            avatar.action = PUT
        }
    }
```

Initialization:

```
    init(denomination)
    {
        self.denomination = denomination
    }
```

Destruction:

```
    smash()
```

Asynchronous actions:

```
    PAID* (amount)
    {
        self.denomination += amount
    }
```

Graphics:

Single static image of a coin. Sound of coins jingling.

Host behavior:

```
    GET
    PAY (targetId, amount) → (success, actualTarget, x, y)
    {
        target = ↑targetId
        if (spend(amount)) {
            if (target.class == AVATAR_CLASS) {
                if (emptyHanded(target)) {
                    createObject(TOKEN_CLASS, target, NULL, HAND)
                    target.inHand.denomination = amount
                    actualTarget = target.inHand.noid
                    x = 0
                    y = 0
                    announceObject(target.inHand)
                } else if (target.inHand.class == TOKEN_CLASS) {
                    target.inHand.denomination += amount
                    target.inHand → PAID* (amount)
                }
            } else {
                newToken = createObject(TOKEN_CLASS, region, target.x, target.y)
                newToken.denomination = amount
                actualTarget = region.noid
                x = target.x
                y = target.y
                announceObject(newToken)
            }
            success = TRUE
        } else {
            success = FALSE
        }
    }
    PUT
```

Object:
towel
Description:
Your basic towel.
Function:
Necessary for every traveler.
Notes:
Everyone knows that this is the single most useful object you can have.
Styles:
None.
Properties:
No properties.
Class properties:
No class properties.
Host properties:
No properties.
Command Behavior:
Do:
`depends()`
Go:
`goTo()`
Stop:
`cease()`
Get:
`goToAndGet()`
Put:
`goToAndDropAt()`
Talk:
`broadcast()`
Reversed Do:
`throw()`
Initialization:
`nullInit()`
Destruction:
`smash()`
Asynchronous actions:
None.
Graphics:
Single static image of towel.
Host behavior:
`GET`
`PUT`
`THROW`

Object:

tree

Description:

Your basic tree.

Function:

Scenic element. Obstruction.

Notes:

This is a fairly inert scenic element, provided almost entirely for visual appeal.

Styles:

```
TREE_CONIFER_LARGE, TREE_CONIFER_MEDIUM, TREE_CONIFER_SMALL, TREE_LEAFY_LARGE,
TREE_LEAFY_MEDIUM, TREE_LEAFY_SMALL, TREE_PALM
```

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(treeStyle)
{
    self.grstyle = treeStyle
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image. Variations possible.

Host behavior:

None.

Object:

truck

Description:

A big truck.

Function:

Carries lots of objects along roads.

Notes:

The truck is a vehicle that can travel on roads. A truck can never travel off the road. In other respects it is like a car, except that it has an enormous cargo capacity.

Styles:

None.

Properties:

```
contents        /* A list of the objects in the truck.  NULL if there are
                   none. */
inhabitants[TRUCK_SIZE] /* The folks riding in it.  There are two slots which
                   are NULL if unoccupied.  The avatar in slot 0 gets to
                   drive. */
moving          /* A flag that the truck is in motion. */
targetX         /* Motion destination X position. */
targetY         /* Motion destination Y position. */
```

Class properties:

```
capacity = ?            /* How many things this can hold. */
displayContents = FALSE /* Whether to show what's in it. */
maxOccupants = TRUCK_SIZE = 2
```

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
climbInOrOut()
```

Stop:

```
cease()
```

Get:

```
goToAndPickFrom()
```

Put:

```
goToAndDropIn()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(moving, targetX, targetY, inhabitants, contentsVector)
{
    self.moving = moving
    self.targetX = targetX
    self.targetY = targetY
    for (i=0; i<TRUCK_SIZE; ++i) {
        self.inhabitants[i] = inhabitants[i]
    }
    unpackContentsVector(self, contentsVector)
}
```

Destruction:

```
smash()
```

Asynchronous actions:

`DRIVE*`

Graphics:

Three static images of truck: side view, front view, back view. Sound of truck engine driving.

Host behavior:

None.

Object:

walkie-talkie

Description:

A radio telephone.

Function:

Like a telephone, but portable.

Notes:

The walkie-talkie operates like a household telephone. The only difference is that it is portable. The receiver is not separate and you have to be holding it in order to operate it.

Styles:

None.

Properties:

```
phoneNumber     /* The 2-byte phone number associated with this phone */
state           /* The state that the phone is currently in.  The possible
                   states that the phone may be in at any given time are
                   PHONE_READY, PHONE_RINGING, PHONE_ACTIVE, PHONE_TALKING,
                   PHONE_LINE_RING and PHONE_LINE_BUSY */
```

Class properties:

No class properties.

Host properties:

```
talker          /* The avatar currently connected to this walkie-talkie. */
caller          /* The phone at the other end of the line. */
beeper          /* Beeper associated with this walkier-talkie.  Always
                   NULL. */
answeringMachine /* Answering machine associated with this walkie-talkie.
                   Always NULL. */
owner           /* Avatar associated with this walkie-talkie.  Always NULL. */
```

Command Behavior:

Do:

```
{
    /* If holding walkie-talkie, hang it up if not already hung up.  Answer
   if ringing.  Otherwise, depends. */
    hangUpOrAnswer(holding(self))
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
{
    /* Answer the phone if it's ringing, otherwise just pick up the
   receiver. */
    answerOrUnhook(goToAndGet(), PHONE_ACTIVE)
}
```

Put:

```
{
    /* If off hook, hang up.  Otherwise drop whatever is in hand next to
   the walkie-talkie. */
    if (holding(self) && (self.state == PHONE_ACTIVE || self.state ==
            PHONE_LINE_BUSY || self.state == PHONE_LING_RING || self.state ==
            PHONE_TALKING)) {
        hangUp()
    } else {
        goToAndDropAt()
```

```
        }
    }
    Talk:
    {
        /* If active, interpret text as a phone call: first text message is
        phone number, further messages are conversation with the person at the
        other end.  When you enter the number, it is dialed.  If there is an
        answer, you can talk.  If not, the walkie-talkie is deactivated when
        you hang up.  If not active, broadcast. */
        dialOrTalk(holding(self))
    }
    Reversed Do:
    throw()
Initialization:
    init(phoneNumber, state)
    {
        self.phoneNumber = phoneNumber
        self.state = state
    }
Destruction:
    smash()
Asynchronous actions:
    ANSWERED*
    ANSWER*
    HANG*
    HUNGUP*
    UNHOOK*
    RING*
    DIAL*
    SPEAK*
```

Graphics:

Single static image of walkie-talkie. Sound of phone ringing. Sound of phone ringing at other end of phone line. Busy signal sound. Dial tone. Dialing sounds (touch tones or rotary clanks). Clicking and clunking when the receiver is picked up or hung up or when the person at the other end hangs up.

Host behavior:

```
ANSWER
DIAL
GET
HANG
PUT
TALK
THROW
UNHOOK
```

Object:

wall

Description:

An interior or exterior wall section.

Function:

Graphic element in buildings. Obstruction.

Notes:

A wall is just an opaque rectangle that sits edgewise on the ground. The edge that sits on the ground defines an impenetrable linear barrier. The only way past a wall is to go through a door (or go around it if it is small, of course). Wall objects are usually (but not exclusively) components of building objects. When they are building components they are not known to the host, functioning as roof objects do. When they stand alone, they *are* known to the host. Walls are painted on the background or foreground as trapezoids.

Styles:

None.

Properties:

```
length          /* How long the wall is.  Its baseline runs 'length' units
                   from the location point in the direction specified by the
                   'orientation' property. */
height          /* How high the wall is. */
orientation     /* What direction the wall runs from the location point, in
                   the form of a number from 0 to 8, where 0 indicates WEST, 1
                   indicates NORTHWEST, 2 indicates NORTH, and so on clockwise
                   around the compass rose.  WEST is always taken as the
                   direction directly away from the viewpoint, even if this
                   is not true West. */
wallPattern     /* The color and texture that the wall appears. */
```

Class properties:

No class properties.

Host properties:

No other properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(length, height, orientation, wallPattern)
{
    self.length = length
    self.height = height
    self.orientation = orientation
    self.wallPattern = wallPattern
}
```

Destruction:
`nullDestroy()`
Asynchronous actions:
None.
Graphics:
Wall image, rendered as one or more texture-mapped trapezoids by the background generator.
Host behavior:
None.

Object:

water

Description:

You know what water is.

Function:

A useful fluid.

Notes:

This is not really an object. It never exists. This entry is just here so that we can say a few words about it. Water is always manipulated indirectly using a container. Certain objects (pond, river, fountain, etc.) are infinite sources of water. From such sources it is possible to fill a water container. The water container can then be emptied onto or into something to have an effect. The water itself is never actually touched or interacted with, so no object need exist to represent it.

Styles:

None.

Properties:

No properties since it is not an object.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

No behavior of its own. Must be manipulated indirectly using a container.

Initialization:

None since it is not an object.

Destruction:

None since it is not an object.

Asynchronous actions:

None.

Graphics:

None.

Host behavior:

None.

Object:

window

Description:

A conventional house window.

Function:

Graphic element in buildings.

Notes:

Window objects are used to decorate the outside of buildings. Like roof objects, they are internal to the home system. They are rather inert.

Styles:

`WINDOW_PICTURE, WINDOW_TWO_PANED, WINDOW_FOUR_PANED, WINDOW_SIX_PANED, WINDOW_GABLED, WINDOW_SMALL, WINDOW_ROUND`

Properties:

No properties.

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
depends()
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
noEffect()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
/* This can't happen */
```

Initialization:

```
init(windowStyle)
{
    self.grstyle = windowStyle
}
```

Destruction:

```
smash()
```

Asynchronous actions:

None.

Graphics:

Single static image for each style of window.

Host behavior:

None.

Object:

wind-up toy

Description:

Your basic windup doll or robot.

Function:

You wind it up and it walks across the floor.

Notes:

The wind-up toy has a key by which you wind it. If you wind it too far it breaks. If you wind it up and then set it down, it walks until it runs down.

Styles:

```
WINDUP_DOLL, WINDUP_ROBOT, WINDUP_PENGUIN
```

Properties:

```
windLevel       /* How wound up this device is.  Normal range is from 0 to 3.
                   Each time you wind it this level is incremented.  If you
                   increment it past 3 it breaks. */
```

Class properties:

No class properties.

Host properties:

No properties.

Command Behavior:

Do:

```
{
    if (holding(self)) {
        self.windLevel++
        @ self!WIND () → ()
        if (self.windLevel == 1) {
            soundEffect(SOUND_WIND_1)
        } else if (self.windLevel == 2) {
            soundEffect(SOUND_WIND_2)
        } else if (self.windLevel == 3) {
            soundEffect(SOUND_WIND_3)
        } else {
            soundEffect(SOUND_SPROING)
            self.windLevel = 4
        }
    } else {
        depends()
    }
}
```

Go:

```
goTo()
```

Stop:

```
cease()
```

Get:

```
goToAndGet()
```

Put:

```
goToAndDropAt()
```

Talk:

```
broadcast()
```

Reversed Do:

```
throw()
```

Initialization:

```
nullInit()
```

Destruction:

```
smash()
```

Asynchronous actions:

```
WIND* ()
{
    self.windLevel++
    if (self.windLevel == 1) {
        soundEffect(SOUND_WIND_1)
    } else if (self.windLevel == 2) {
        soundEffect(SOUND_WIND_2)
    } else if (self.windLevel == 3) {
        soundEffect(SOUND_WIND_3)
    } else {
        soundEffect(SOUND_SPROING)
        self.windLevel = 4
    }
}
```

Graphics:

Single static image of wind-up toy for each style.

Host behavior:

```
GET
PUT   /* *** Need special put to handle toy walking away. *** */
THROW
WIND () → ()
{
    if (holding(self)) {
        self.windLevel++
        if (self.windLevel > 4) {
            self.windLevel = 4
        }
        # self → WIND* ()
    }
}
```
