Jefferson TDR

Chapter 4 : People
Jamie Doornbos
10/05/97
Revision 1.

Chapter 4 : People

People in Jefferson are represented by class cXPerson, a subclass of cXObject. Many of the processes and resources for people are the same as those for objects. Most notably, the graphics, simulation, and animation are different. The graphics and animation are hooked into vitaboy technology using class XVitaBoy and XAnimator. The simulation has the same behavior tree component as objects along with a motive simulation managed by class Motives.

4.1 Type Resources

4.1.1 'STR#' or 'CST', Body Strings

In order to use instantiate vitaboy, names of skeletons and suits and registration points must be provided to the vitaboy character animation module. These values are configured differently for different people, and are stored by class BodyStrings. The class BodyStrings is subclassed from StringSet, which is just an array of strings saved in a resource. BodyStrings contains an enumeration to define what each string in the set refers to:

```
enum {
        kSkeletonName=1,        // skeleton that the person will use
        kSuit1Name,             // basic suit, or complete suit
        kSuit2Name,             // additional suit (accessories?)
        kHeadName,              // the head bone
        kHeadRegName,                       // the head reg pt
        kRightHandName,         // the right hand bone
        kRightHandRegName,      // the right hand reg. point name

        kDefaultFaceName,
        kHappyFaceName,
        kSadFaceName,
        kMadFaceName,
        kSleepFaceName
};
```

TBD: more strings may need to go in here, such as accessory suit names, hair styles, and colors. TBD: this resource may be editable when the user is creating a new person and changing its appearance.
Each instance of person has an instance of body strings, which is initialized when the person is initialized. TBD: the body strings may as well be in the type structure, ObjSelector, since they do not change for instances. The body strings are passed in to XVitaBoy::MakeNewVitaBoy to generate a new XVitaBoy object and are passed in the constructor for an XAnimator object.

4.1.2 'pers', Personality

The class Personality keeps a static array of weights, fWeights. Personality is currently not really used by anything but is still stored in each person and loaded when the person is constructed. TBD TODO: it should be loaded when initialized instead. TBD: This class may be revived or scavenged for future personality implementation.
The array entries from 0 to 55 correspond to the tree table advertisements, 2 for each advertisement, of which there are 28. TBD TODO: this number is stale and should be fixed to use the same number as in class Motives: NUMMOTIVES=16. The reason for this is that the tree table advertisements used to correspond to the old motive implementation, a "need" array. The first of the 2 values is the multiplier for

incrementing the need, and the second is the multiplier is for decrementing a need. Entries higher than 55 are currently ignored. Personality::GetWeightName takes an array index and returns a string representing the weight name. The routine currently returns an empty string for values greater than 55, Otherwise, it uses a string resource from the language file (see Virtual Machine section ???) to get the motive name and appends “inc” or “dec” depending on the parity of the index.
The values for a person’s weights are stored in a pers resource, which is just a null-terminated string containing name-value pairs separated by white space, such as this:
“hunger inc” = 1.6 “hunger dec” = 0.8
The values are floating point numbers which are entered into the weight table by matching the name against the weight names returned by GetWeightName. This string based specification of weight values is only used when loading the personality. The weights are subsequently referenced by array position. The string-based specification is used to let the ‘pers’ resource specify only non-default values.

4.1.3 Graphics. See also section on character animation ???

Animations, skeletons and suits are stored in a global cmx folder. The names of a person’s skeletons and suits are specified in the body strings (see above section 4.1.1) and are expected to reside in this folder. The textures referenced by the suits are stored in a global Textures folder. TBD: if and how these resources should be localized to the person. The names of skills are stored in the person’s Animation Table, and are then referenced by their index in this table. The animation table of a person has predefined animation types for some indices, referred to as stock animations. These are enumerated in class XAnimator:

```
enum {
        kRest,
        kWalk,
        kRightHandCarry,
        kHunger,
        kSleepy,
        kWired,
        kBladder,
        kMad,
        kEntertained,
        kNumAnimTypes
};
```

Thus a behavior tree can call out stock animation number 4 on any person, and that person will do his version of being sleepy. See also the section on Animation Tables, 3.2.9.
Class VBModule is where vitaboy globals are stored:

- animation manager, an instance of VBAnimMgr
- pointer to a 3D device used for rendering vitaboys
- pointers to two selection lights for rendering selected vitaboys

A static instance of this class is contained in class XVitaBoy, and gets initialized from XVitaBoy::Init and destroyed from XVitaBoy::Destroy. Upon initialization, VBModule sets up the animation manager and cmx resource site, and loads all animations. It also sets the skeleton event handler to a locally defined function which just looks for “xevt” event keys, casts the call-back data to a cXObject and calls it’s handling routine with the event number. The call-back data is set before each animation tick to be the pointer to the person whose animation is being ticked.

4.2 Family Membership

A person is a member of a family if the data field, fData[kFamilyNumber] is non-zero. If the number in that field matches the global simulation data field, kCurrentFamily, then the person lives in the current house being played. TBD: can more than one family live in a “house” (house file)? Also, if the person lives in the current house, it can be controlled by the user and appears in the family window. Currently there is no user interface for setting up a family. TBD: how to set up a family? In order to have a family, there needs to be a family name, a family number, and some people that have that number as their family number. TBD: People not yet in houses and/or families will probably be stored in a special house file that does not appear on the neighborhood screen, such as “House19.iff”.

4.4 Personalities

As a working definition, personality is a set of parameters that affect the changes in motives caused by different conditions, including the passing of time. The motive engine is currently not using such parameters. TBD: how to implement personality.

4.4.1 Effects

4.4.1.1 Motive increment factors

TBD: For each motive, there should be two personality parameters, the increase multiplier and the decrease multiplier. The parameters are used to multiply the operand each time a behavior tree uses "+=" or "-=" on a motive. Currently these multipliers are all implicitly 1.0.

4.4.1.2 Other simulation coefficients

TBD: other parameters in the personality may be how different motives interact. For example, how much low hygiene affects comfort, or how boredom affects alertness, or the multiplier for cubic limiting of comfort, etc. Currently such parameters are hard-wired into the motive simulation.

4.4.2 Editing (Custom Characters)

Each personality value will have a label, which will be used by the personality editor to edit the value. TBD: May also need statically defined minimum and maximum for each value if the values are not modulated to fit in a constant range, like 0 to 100. TBD: The resource used to store the values may be similar to the one described above in section 4.1.2.

4.4.3 Evolution/Decay under simulation

The most straight forward personality implementation is just an array of parameters such as the ones described above in section 4.4.1. The problem with this is that it is very difficult for an object to have lasting effects on a person. A way of defining the changes in a personality factor would bring a lot of functionality into the system.

TBD DESIGN IDEA: One idea for implementing time-variant personality factors. Let a personality be an enumeration of multipliers for different motive effects. Some such multipliers may include "filling bladder per tick" and "hunger increment per tick", or "unit happiness decrease". These multipliers would be used by the motive engine to modify motive effects as necessary. Each multiplier would have a base value and a series of envelopes. The envelopes would be parameterized by shape, delay, duration, and amplitude. The final value is determined by adding all the envelope values to the base value.

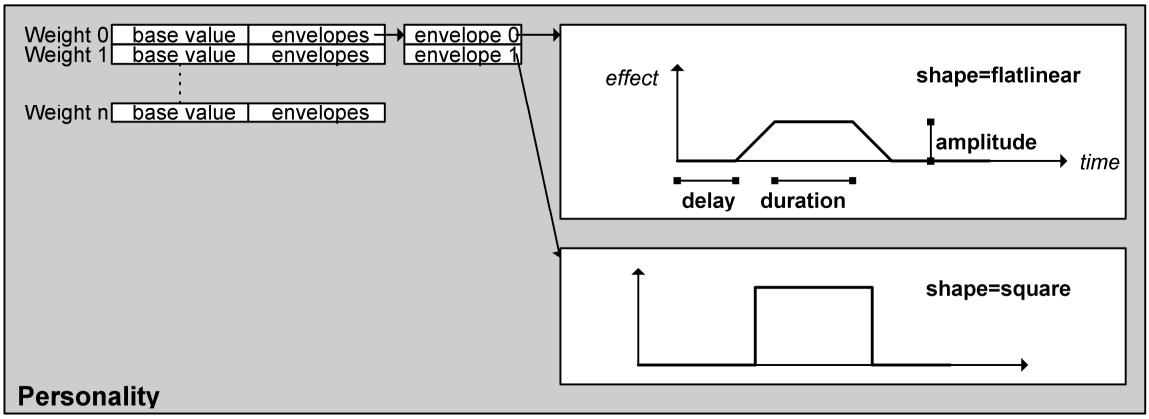

Figure: Sample picture of a proposed Personality structure with two active envelopes on Weight 0.

Once the lifetime of an envelope is over, it would be removed from the series. A behavior primitive would exist for adding an envelope to the series, and possibly for checking and removing envelopes. This would enable simulation effects like "sickness causes depression" or "drinking coffee causes a full bladder one

hour later". TBD: Permanent alteration of the base value, or trauma modeling. If the total additive values of the envelopes are above a certain threshold, the base value could be altered. This would require the base values to persist with the person.

TBD DESIGN IDEA 2: Effect trees. Embed the personality effects in behavior trees that are attached to a person and run in addition to the person's main tree. Interaction trees would use a primitive to attach effect trees to a person, which would run to completion and be deleted. The effect trees could implement the envelopes described above, or any other function that can be represented in primitives. The personality envelope trees would be understood to have a zero net effect over their duration, unless a permanent alteration is deemed appropriate. The implementation in C++ would be quite simple: just a list of trees that run on the person's sim tick and get saved with the person. Global effect trees that take delay, duration, and amplitude as parameters could make the invocation of an effect tree envelope almost as easy as it would be with an envelope primitive.

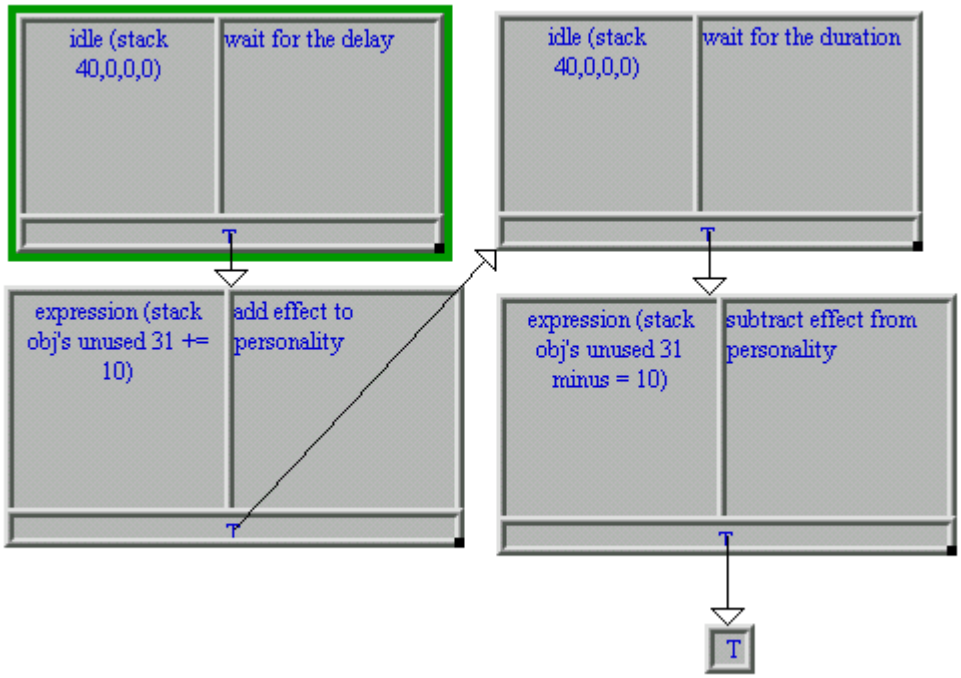

Figure: how a personality effect tree might look. This tree demonstrates a square envelope effect similar to that described in the previous figure. "unused 31" is used to represent a personality parameter, since none currently exist in the behavior editor.

4.5 Relationships

4.5.1 Data Structure

The data structure for relationships is a collection of dynamically sized arrays. It is commonly referred to as a *relationship matrix*. Each array is indexed by an integer value, the key for the array. A key and an index are needed to get a value. A key, index, and value are needed to set a value. An array for a given key is not present until its size is set to a positive value. The structure can be thought of as a sparse array, where the keys index the rows and the array indices index the columns. Only rows for keys with arrays of length greater than zero are allocated, and only columns from 0 to the length minus 1 are allocated in each of these rows:

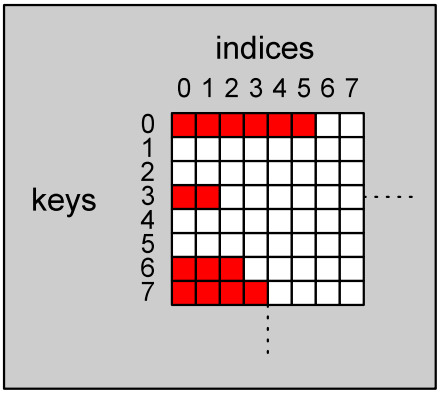

Figure: a relationship matrix with 4 active keys shown as a sparse array. Colored squares show allocated memory.

4.5.2 Among People

For person to person relationships, each person instance has a relationship matrix. The keys are the object ids of other people in the world. The indices of each array correspond to parameters governing how a person feels about another. The quantity for each index may be a range value or one of a set of enumerated constants. The quantities in the array are the specific values for how the person containing the matrix relates to the person with the object id of the array's key. TBD: what are the interpretations for each index?

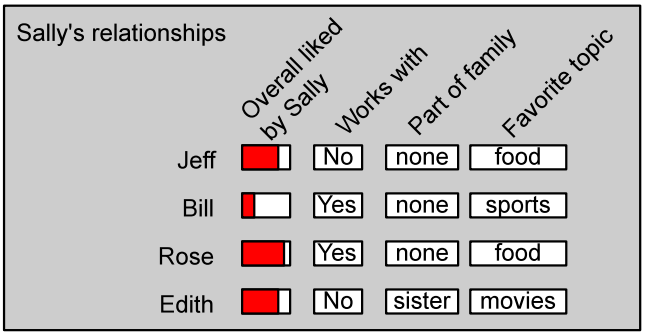

Figure: how a sample person-to-person relationship matrix might look.

4.5.3 Object instances

The relationship of a person to an object is stored in the same matrix as that of the person to other people. The keys, as before, are the object ids of the objects that the person has a relationship to. The indices and quantities correspond to different things depending on which object the relationship is for. TBD: The interpretation of the indices as well as how each interaction changes them should be part of the written document that specifies an object. In the interaction tree for an object, primitives are used to see if the relationship exists, define how many quantities should be in the relationship, what the initial values are, and how they are affected by an interaction. See also "relationship variable" primitive, section ???.

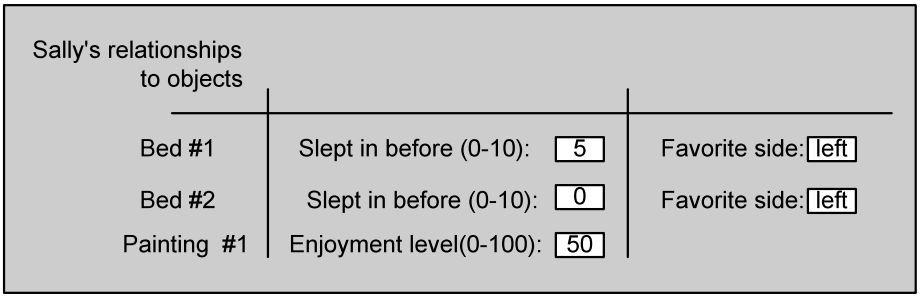

Figure: how a matrix with relationships to some sample objects might look.

4.5.4 TBD: Affinities

For a person's relationship to generic, global, groups of properties, each person may have a relationship matrix where the keys are "affinities" and the indices correspond to different characteristics of the affinity. For example, one affinity may be music. In the array for music, the indices may correspond to "overall talent", "favorite type", "second favorite type", "hated type". Both a radio and a guitar may reference a person's affinity matrix to determine the effect of their interactions. Each affinity is a unique integer generated like a globally unique identifier. A table of known affinities would be stored, with labels, in editor resources or code. Registering an affinity would amount to entering it into the table. [TBD: If affinities are displayed anywhere in the game and plug-in objects are allowed to bring in new affinities, registration of new affinities and labels will be needed. This can be easily accomplished by storing the table of known affinities as a resource and importing such resources of plug-in objects.] The labels of known affinities would show up in edith in the as yet undefined "affinity variable" primitive, akin to the "relationship variable" primitive. Any affinity key can be referenced, however, so that new ones may be added after shipping. The affinity matrix provides a way for objects to communicate their effects through a group of related properties. TBD: the affinity matrix concept bears strong resemblance to personality, but can accommodate new affinities from plug-in objects.

4.5.5 Object Types

Relationships to object types are a special case of affinities, where the unique integer affinity is just the guid of the object type.

4.6 Instance data

4.6.1 Location and orientation

A person has additional high-precision location information in addition to that of regular objects. This is so that a person can follow a straight line with precise slope and no stair stepping effects.

4.6.1.1 Regular World tile and object direction

These are the same as for objects and consist of a world tile coordinate for the origin, and an integer from 0 to 7 for the direction (chapter 3, section 3.3.2). The route following code sets these by rounding the higher precision coordinates described below to the nearest world tile coordinate (chapter 2, section 2.1.1.1) and object direction (section 2.1.3).

4.6.1.2 High precision world tile and object direction.

In addition to the normal object location and direction, a person also includes a floating point representation of the same. They are stored in 3 member variables, one for each coordinate: x, y, and rotation. The values are only set by the route following code and are only used when the person is following a route.

4.6.2 XVitaBoy

People are drawn using vitaboy 3D animated characters. The animations are stored as hierarchical transformations on a set of bones. See also section on Character Animation, ???. Each person has an instance of class XVitaBoy and of class XAnimator to manage the body and motion of the underlying Vitaboy object.

4.6.2.1 Root Position

The XVitaBoy class stores the transform which gets applied to the root of vitaboy's meshes. On each simulation tick, the transform is computed using the location and direction of the person. XVitaBoy provides 2 SetLocation routines for doing this, one for regular object coordinates and one for high precision. If the person is following a route, the high precision coordinates are used to call SetLocation, and if not, the regular coordinates are used. SetLocation uses global coordinate conversion functions to go from isometric coordinates and object rotation to world coordinates and degrees, and finally to a transform (see 3D graphics, section ???).

4.6.2.2 Registration points

Registration objects are part of the vita boy interface and are used to get the world coordinates of a particular part of the skeleton. Three registration points are set up by XVitaboy when it is initialized, the hand registration and two head registrations. The hand registration is used to draw the object that the person is carrying. The head registrations are used to draw balloons, icons, and headlines. In code, there is no difference among these; they are just different sprites.

4.6.2.3 Flashing timers

Currently people are highlighted using a flashing mechanism. The XVitaBoy contains a timer for determining when the flash should be turned on and off. Each time it is rendered the timer is checked, and if expired, the brightness is toggled between normal and bright. TBD: the highlighting scheme may change and these fields might go away.

4.6.3 XAnimator

Each person has an instance of XAnimator which provides a simple interface for handling vitaboy animations. It implements animation channels so that a person can be running several animations at once. XAnimator collects animation events in a member queue structure, which is then accessed by the animation primitive to run an animation event tree (see "Animate New" primitive, section ???). XAnimator keeps track of a person's faces, and provides an interface for setting them for a particular number of ticks.

4.6.3.1 Animation Channels

When an animation is set to a person, a channel is also provided. Different channels cause different effects on the overall state of the person's animation.

Implemented Effects

TBD: do we need any other animation effects and what are they?

1. Repeat Mode: loop or hold

Specifies what happens at the end of an animation. Loop causes the animation to repeat. Hold causes it to stop at the last frame.

2. Fading: true or false

Specifies if the animation should be mixed in over a certain amount of time. The amount of time is a constant set to half of a second. TBD: can this remain a constant? TBD: this constant should go into tweaking screen.

3. Exclusivity with other channels

Setting the animation on some channels will automatically stop animation on other channels.

4. Opaque: true or false

Setting the animation on some channels will block other channels using the opaque feature.

Channels

Each channel runs one animation at a time. A channel is currently just a vitaboy Practice. TBD: channels may need to be beefed up with more information. Any time an animation is set to a channel, any old animation in that channel is stopped, in addition to effects listed below. TBD: are more channels needed?

1.Background

Repeat mode: looped.
Fading: true unless the travelling channel is active.
Exclusive: stops object and travelling animations.

Opaque: false.

2. Travel

Repeat Mode: looped.
Fading: false.
Exclusive: stops background animation.
Opaque: false.
The travel channel also sets the scale of the animation to a value other than 1.0. XAnimator provides a function for setting the traveling speed, which is simply cached and later used with the natural speed of traveling animations to scale them and prevent sliding. TBD: this does not appear to work—why? Currently the travel speed is set exclusively from the person's SetSpeed function, which is called whenever the person begins following a route. TBD: this may become the inverse, i.e. the natural speed of a traveling animation will determine the person's travel speed.

3. Object

Repeat Mode: hold.
Fading: true unless the object channel is active.
Exclusive: none.
Opaque: true.

4. Carry

TBD: carry channel is not really fleshed out, as there are no carry animations.

5. Motive

Repeat Mode: hold.
Fading: true.
Exclusive: none.
Opaque: true.
In addition, the motive channel automatically stops when it is finished, and other active channels are set to fade in.

Example : Eating while sitting.

TODO: put example here.

4.6.3.2 Events

When an animation event happens, the cXObject routine HandleAnimEvent is called from XVitaBoy. This routine puts the event number into the lapsed events queue of the animator of the person whose animation caused the event. TBD TODO: HandleAnimEvent should be moved to XAnimator for encapsulation.

4.6.3.2 Faces

XAnimator is initialized with a BodyStings object, which contains face names for happy, sad, etc. These face names are used to initialize a set of suits that are the faces. A function is provided to set the face for a given number of sim ticks. The XAnimator takes care of dressing the suit onto the vitaboy and returning it to normal after the ticks have expired.

4.6.4 Motives

The class Motives, authored by Will, encapsulates a person's motive simulation, and each person has a direct member variable which is an instance of this class. The basic interface to the class consists of an Init function and a Sim function. There are special call backs also declared in the class that allow graphical side effects to be called out when motive threshold values are crossed.

Simulation

The person's main simulation, which is called every sim tick, calls the motive simulation at a fixed sim-tick interval, determined by the constant cXPerson :: kSimTicksPerMotiveTick, which is 20. The motives class stores two copies of the person's 16 floating point motives, an old one and a new one. The sim uses the difference between the old and new to calculate different effects. At the end of the motive simulation, the old one is copied into the new one. TBD: the motive simulation should take into account personality effects and possibly tweakables. TBD: should the motive sim affect the personality?

Graphical Side Effects

In the Motives .h file, the motive constants are enumerated, as well as some enumerations for faces, icons, animations, and headlines. Functions for setting these person display properties are declared as members in the Motives class. The functions are conditionally defined in person.cpp for tds and in motive.cpp for non-tds applications. This is so that the Motives class can easily plug in to the motive test application authored by Will. The functions in person.cpp utilize a cXPerson pointer stashed in the motives at init time to affect the UI change requested.

4.6.5 Route following

A list of points and the current point being walked to are stored in the person. When a person is following a route, these are used to move the person to its next location. The list is only valid during route following. See also Movement, Chapter 9.

4.6.6 Action Queue

A person has an action queue of things that it should do when it gets a chance. An action consists of a stack object id and an action number. The stack object id refers to the object that should be interacted with (as well as the object that will become the stack object during the interaction). The action number refers to the index of the action in the stack object's table of interactions (see Tree Tables, section 3.2.6). Also see "Idle For Input" primitive in virtual machine chapter ???.

4.6.7 Personality

Each person has an instance of class Personality. Currently this class is not used for anything, but probably will be. TBD: what is personality?

4.6.8 TBD: Name

Instances of people will probably be able to be renamed, so a new member will be needed to store the name. TBD: this field could go into the base object class if objects can also be renamed.

4.7 TBD: Skills

A person will have a set number of skill factors that represent how good the person is at different activities. Interaction trees will access the relevant skills to determine the person's efficiency at the interactions, and the results produced.
TODO: insert the current list of skills.

4.8 Off-screen people

TBD TODO: One-time visitors

Visitors that provide special interactions on an as-needed basis and do not maintain relationships with the members of the family will be managed by a special visitor creator object or a set of such objects. These objects will implement heuristics for when a visitor needs to be created and which one to create. Such visitors may include salesmen, doctors, undertakers, and mail carriers. The visitor creator object will create the visitor and possibly direct their initial action, and set a relationship flag so that they can be called back and deleted. TBD: protocol for visitor-creator communications.

TBD: Persistent visitors—Friends

TBD: When a visitor comes in, how is their persistence managed? A relationship to a friend should remain the same between visits, but the resources (memory and processor) consumed by a person may be too great to have the friends be full-blown cXPerson objects all the time.

TBD: Some relevant questions are:

- Can a visitor object type have more than one instance active at a time?
- How is the set of available friends for a house (or family or family member) determined?

TBD POSSIBILITY 1: A fairly straightforward solution is to deallocate everything possible when a friend is not on screen, keeping their cXPerson instance allocated, but in a very disabled state, possibly even removed from the object list to not bog down searches. The person's simulation would be disabled as well. TBD: a method for "waking" a disabled object would be needed for this.

TBD POSSIBILITY 2: Another way of solving this problem may be to have a special storage area for relationships of deleted objects. When a friend is deleted, his relationship matrices would be transferred to a global memory area, and a unique integer generated that could be stored and later used to re-attach the matrices. Perhaps a friend creator object, similar to the visitor creator, could manage the process of storing the relationship identifier and reviving the friend at a later time. In primitive form this may amount to a "clear/restore relationships" primitive which on clear, returns a number, and takes a number to restore.

TBD POSSIBILITY 3: Save the person in a file and delete them. First, when the friend is done visiting, he would be "turned off", a process that makes sure the friend is sufficiently isolated from the house environment. Then he would be saved in a file, which may be the same house file that he is visiting, or another special house file. Before deleting the friend, his object id would be reserved for later use when he is called on again.

TBD: Neighbors

TBD: How to get people from other house files to come in like visitors. TBD: Do neighbors need to have persistent relationships with family members? If so, this is quite similar to the friend problem, except the targeted object type is one from a different house. TBD: Propagating the changes in a neighbor back to the original house will not be implemented.