Jefferson TDR

Section 3 : Objects
Jamie Doornbos
8/27/97
Revision 0.

Introduction

3.1 Types and Instances

For each object type, there is one and only one corresponding OBJD resource in a file in the Objects folder. Various routines in ObjectSelector.cpp take care of finding all the types and for each one, building an 'ObjSelector' struct, which contains certain static resources for the type. The class 'ObjectModule' is a container for instances. It knows how to turn an 'ObjSelector' into a 'cXObject' (including subclasses).

3.2 Type Resourcess

There are a multitude of things to be specified for instantiating an object, as well as for displaying and interacting with the object. In Jefferson, these are consolidated in OBJD resources on disk, and in selectors at runtime. By the time everything is dereferenced properly, there will be an object with graphics, behavior, sounds and animations.

3.2.1 OBJD Resources

An OBJD resource contains all the information necessary to construct and initialize a runtime instance of the object (except for the semi-global file, see 3.1.2). It is just an array of integer values with various interpretations. The resource is edited (by name-value pairs) in edith in the Object Definition window. The end of the data is padded with ample zeroes to allow for future additions without hassle of version updating (as long as zero is permissible value for new fields).

3.2.1.1 version – the current version of OBJD. Has not changed for a while because the zero padding trick.

3.2.1.2 stackSize – how many levels are allocated in the tree simulation's stack for this object. TBD: this may end up being more dynamic.

3.2.1.3 baseGraphic – the id of the DGRP resource which corresponds to graphic number 0.

3.2.1.4 numGraphics – the supposed number of DGRP resource allowed to be used by this object.

3.2.1.5 initBhav – the id of the initial tree the object is to execute. This tree is called outside of the regular simulation to initialize the object. This is necessary because some important fields are needed while the object is being placed, and the sim for the object is disabled during placement. Currently the initialization is assumed to take place in the first tick of the init tree, and the rest is the real 'main' tree. TBD: init tree and main tree may need to be separated so the init can be run separately without disturbing current execution state.

3.2.1.6 toolbarPict – the id of the PICT resource used to draw an icon of the object in the pull down object menu. This is not used, as we currently have text in that menu.

3.2.1.7 treeTableID – the id of the TTAB resource which is the interaction table for this object.

3.2.1.8 personalityID – the id of the PERS resource which is the personality for this object. This field is ignored for non-people types.

3.2.1.9 type – one of several values: kUnknown, kFood, kPerson, kContainer, kFurniture, kStructure, kAnimal, kSimType, kDoor, kMouseType, kUserAvatar, kInternal. The type field is used to determine the C++ type of object to construct. Currently, all of these types map to cXObject except for: kUserAvatar maps to cXUserAvatar (not used); kDoor maps to cXDoor; kMouseType maps to cXMTObject(multi-tile object); kPerson maps to cXPerson. It is also used by different modules to determine some special properties.

3.2.1.10 masterID – the id that binds multi-tile objects together. If zero, object is single-tile.

3.2.1.11 subIndex – this is an encoded x,y offset for multi-tile objects. It is ignored for single-tile objects. The top 8 bits (most significant) are the x offset and the bottom 8 are the y offset. Both x and y must be greater than or equal to zero. If subIndex==-1, then this is the master type for the multitile object. When the object module is requested to instantiate a master type, it actually instantiates all the other types with the same master id, and links them up (compilcated piece of code). Requesting an instance of a part of a multi-tile object (masterID!=0 and subIndex!=-1) is an error. TBD: these are hard to type, separate x and y fields needed. The mapping to and from master and sub objects is isolated in global functions GetSubTileSelector and GetMasterSelector in ObjSelector.cpp.

3.2.1.12 dialogID – the id of the DITL resource to display when the user requests it. This is not used anymore. TBD: may be reinstated if we decide to use dialogs.

3.2.1.13 animTableID – the id of the STR# resource (or CST) that this object uses as its animation table. No animation table is loaded if this is zero.

3.2.1.14 guid – Globally unique identifier. Needed for saving and loading objects. Generated by code in StubObject.cpp, called by TDSBuildObject when used with -tmpl option.

3.2.1.15 disabled – can be set in the editor to keep an OBJD from being used in the game.

3.2.1.16 portalTreeID – for door objects, the id of the behavior tree to run when someone needs to walk through the door. Ignored for non-doors.

3.2.1.17 price – cost of the object. TBD: should this be dynamic or at least transferred to dynamic fields?

3.2.1.18 bodyStringsID – for people, the id of the STR# resource to use as the body strings. Strings in this string group correspond to an enum which specifies which suits and skeletons a person uses, and also reg. points, etc. Ignored for non-people types.

3.2.1.19 slotsID – if of SLOT resource to use for this objects slots. A slot resource is a list of binary slots.

3.2.1.20 headLinesID – the base id of the SPR# resources to use for this objects imported headlines. A behavior tree can set the headline to a local headline index. Index 0 corresponds to this sprite.

3.2.1.21 eventTreeID – id of behavior tree to be run when an animation event occurs. No longer used.

3.2.1.22 selfModTreeTableID – id of TTAB resource used as the reflexive interaction table. The trees in this table appear on a menu when the object is clicked on with the reflexive tree tool.

3.2.2 GLOB resources

The GLOB resource is just a string (length prefixed) that specifies the name of the semi-global file to use when instantiating a Behavior for this object. This is expected to be relative to the Global directory. A semi-gobal file contains behavior trees with ids in the semi-global range. The GLOB resource is one-per-file. Having more than one will cause an undetermined one of them to be ignored. TBD: Does a GLOB resource need to be specified in each object definition?

3.2.3 BHAV : Behavior Trees

The BHAV resource is the lean and mean array of instructions. Its resource id is also the “tree id” and is used to reference the resource from a variety of places. These resources are stored in possibly 3 files for each object. The file for a given tree id is determined by the range it is contained in. This id-to-file mapping is done in class Behavior and class LayerBehavior. Class Behavior also takes care of locking down all the resources and parsing the array to return a simple instruction line. (see also chapter 7 ???).

3.2.4 POSI : Positional resources

The POSI resource is the saved state of the editor window for a BHAV. It contains the position and size of each node in the editing window, as well as comments, and extra nodes that do not appear in the behavior itself. For a given BHAV resource, the corresponding POSI resource is just the one in the same file with the same id. POSI resources are only loaded when a tree is to be viewed, such as in trace mode, or in edith.

3.2.5 Selectors

An ObjSelector is the runtime struct for stashing various type-related data which do not change over the span of the game, including a pointer to the OBJD resource and iResFile objects. To build the table of all selectors, each file in the Objects folder is analyzed for OBJD resources. For each OBJD resource, a selector is built and added to the table. Thus selectors are one-to-one with OBJD resources (execpt for disabled types, see 3.1.1.15).

To build a selector, 3 iResFile objects are created:

private, where the OBJD came from;
semi-gloal, see 3.2.2, can be NULL; and
global, the same for all objects : TDSScen/Global/Global

A cRenderer object is made from the private file.

A LayerBehavior object is made from all 3 files.

An Animtable is loaded if the animTableID field of the OBJD is non-zero (see 3.2.1.13).

The name of the OBJD resource is cached as the object's name. TBD: How to name people and objects for real.

3.2.6 TTAB : Tree Tables

A tree table stores a variable number of interactions. An interaction consists of a check tree id and an action tree id and satisfaction levels for each motive. The satisfaction levels are compressed to only give the non-zero values in the resource. At runtime, these resource are loaded by a ResTreeTab class. A subclass, ObjTreeTab, is contained in each object instance. Each object instance gets its own copy of the static tree table for the type, though currently the values are not saved. TBD: Currently, each object has a copy of the type's tree table, but do they really need it? (see also: interactions ???)

3.2.7 SPR# : Sprite List

A single sprite is identified by its file, its resource id, and its index. An SPR# resource contains version info and an offset table. An individual sprite is found using this table to add to the address of the beginning of a resource. Each sprite has header info and compression data. The header has a bit mask denoting which types of tokens it contains, and a bounding box. The compression data is tokenized to do run length, clear space, patterns, and shades. (see also ???).

3.2.8 DGRP : Draw Groups

A draw group contains enough information to render a single frame of an object in any of 4 rotations and 3 zooms. This is done with a list of draw lists. Each draw list has a bit mask denoting which rotation the frame is valid for, and an integer value denoting the zoom. To draw an object, its rotation is is added to the view rotation, and then, with the current viewing zoom, the appropriate draw list is selected. Within each draw list is a list of tokenized "draw items". The different types of tokens that can occur are:

1. Sprite: id and index with x,y pixel offset and flipping flags.
2. Top Object: x, y offset of where to draw the target object (not used anymore)
3. Body Token: draw this object's body (not used anymore)
4. Slot Object Token : (x, y, alt) 3D fixed-point coordinate in object's frame of reference. A slot index denoting which slot object to draw there.

Slot Object Tokens can currently be left out of the draw group entirely, and the slot objects will be drawn from the slot list itself. (See also Slots,???)

3.2.9 STR# (or cst) : Animation Table

STR# is a list of strings that refer to vitaboy skill names. The animation table is loaded from the three resource files of an object by taking skill names from the private file first, then filling in any blanks with the semi-global and then the global. TBD: do we really need the cascaded animation tables? It seems like objects will either know what their animation is or not. The person's animation table has standard animations that all people have, such as stand and sit. In this way, an object can tell a person to do their own special categorical animation without knowing what it is. (See also Animate New:???) An object's animation table contains animations for interacting with that object. Currently, no objects have their own animations. All are stored in the people. [TBD: How to get animations for multiple skeletons transparently into the object's animation list. This can be done by having a new skeleton parameter in the animation table's GetEntry routine. Then to load an animation table, a resource id for each skeleton type must be provided, which would just be new fields in the OBJD.]

3.3 Instance Data

Basically these ar the members of the cXObject class

3.3.1 Properties. Members that are the same for the lifetime of the object.

- **3.3.1.1** ObjectModule* fModule – the ObjectModule that this object belongs to.
- **3.3.1.2** cXObject* fNext – next object in the module's object list
- **3.3.1.3** Sint16 fID – the id of the object. Used in the world grid (see World, chapter 2), and in the module's object table. Ranges between 1 and ObjectModule::kMaxObjects. ID 0 is null.
- **3.3.1.4** Definition* fDef – the object definition that was used to create this object. This is a pointer to an OBJD resource which has been locked down.
- **3.3.1.5** ObjTreeTab* fTreeTable – the tree table of this object. Used when a person autonomously looks for interactions or when the user clicks to see a menu. The trees in this table are assumed to be run by the interacting person with the targeted object in the stack object field (see also Virtual Machine: ???). The advertisements in the table can be modified by the behavior trees, but are not currently being written to disk. TBD: No current objects are statically modifying tree table advertisements, should it be allowed?
- **3.3.1.6** ResTreeTab* fSelfModTreeTab – the reflexive tree table of this object. Use when the user clicks on an object with the reflexive tree tool. The trees in this table are assumed to be run by the object.
- **3.3.1.7** ObjSelector* fObjSel – the selector used to create this object.
- **3.3.1.8** cVoice* fVoice, cChannel* fChannel – Formerly used in Jacques' feature: X People talking on the mac. Now is not used and stubbed out.
- **3.3.1.9** RTSlotList fSlots – the list of slots for this object (see also Slots: ???).

3.3.2 State. Variables that change through simulation or other means.

- **3.3.2.1** FTileRect fRect – the rectangle of the object in fixed world fractional tile coordinates.
- **3.3.2.2** FTilePt fLocation – the location of the object. Always at the center of fTileRect
- **3.3.2.3** RelMatrix fInstMatrix – the relationship matrix keyed off of the ids of people in the module
- **3.3.2.4** Sint16 fSimEnabled – whether or not the object is simulated. Currently turned off only during placement.
- **3.3.2.5** Sint16 fData[kNumData] – Data Array. Currently kNumData is 36. Accessed directly by behavior trees and compiled code. An asterisk(*) means the value is only used in people.
  - 3.3.2.5.1 kGraphicNumber – the current graphical frame the object has (0..numGraphics-1)
  - 3.3.2.5.2 kDirection – the current direction the object is facing (0-7)
  - 3.3.2.5.3 kColor1 and kColor2 – the palette entries to use when blitting a shaded sprite as part of the object. TBD: do we need shaded sprites?
  - 3.3.2.5.4 kPattern – the pattern number to use when blitting a patterned sprite as part of the object. TBD: do we need patterned sprites?
  - 3.3.2.5.5 kHeight – the nominal height of the object. Was used to determine where to draw the hilite arrow above objects, but currently not used.
  - 3.3.2.5.6 kRouteID* – the id of the route that a person is following.
  - 3.3.2.5.7 kIndirectID – previously used in some primitives for accessing fields in another object. Currently not used.
  - 3.3.2.5.8 kFlagField1 – set of various flags for the object.
    - 3.3.2.5.8.1 kCanWalkOver – whether or not an object can be stepped on as though it were the ground.

        - 3.3.2.5.8.2 kCanWalk – whether or not the object is mobile. Should be set for all people all the time.
        - 3.3.2.5.8.3 kPreviouslyFound – whether or not the object has been found in a behavior tree search. Cleared before starting a search. Field is not guaranteed after a tick has passed.
        - 3.3.2.5.8.4 kOccupied – whether or not to allow interaction with the object. Set by behavior trees. Tested by code and trees.
        - 3.3.2.5.8.5 kNotified* – Whether or not the user clicked when the person was in a non-interruptable Idle For Input primitive (See also Idle For Input Primitive, ???).
        - 3.3.2.5.8.6 kRoutingInterruptable* – Whether or not the goto relative primitive will fail immediately if an action is in the action queue. TBD: Can be used to force a routing action to terminate. (See also Goto Relative: ???)
    - 3.3.2.5.9 kAnimID* – Currently is always 0 or 1. 0 means no object animation is playing. 1 means that it is (see also Primitive Animate New: ???). Formerly to be used (and is still in #ifdefs) for the resource id of the animation being played on the old-style person bodies.
    - 3.3.2.5.10 kAnimFrame – Currently not used. Formerly used in the old style body animation to be the current frame.
    - 3.3.2.5.11 kObjectID – the id of the object. This is a duplicate of member fID so behavior trees can access it.
    - 3.3.2.5.12 kOldTargetID – not used directly. Only here as a placeholder for behavior tree references. Mapped by primitives to the object id in the default slot (slot 0).
    - 3.3.2.5.13 kWallPlacementFlags – set of flags for where the object can be legally placed with respect to walls.
        - 3.3.2.5.13.1 kWFAnywhere – walls do not affect placement.
        - 3.3.2.5.13.2 kWFInFront – wall required in fron of object
        - 3.3.2.5.13.3 kWFRightSide – wall required on right side of object
        - 3.3.2.5.13.4 kWFBehind – wall required behind object
        - 3.3.2.5.13.5 kWFLeftSide – wall required on left side of object
        - 3.3.2.5.13.6 kWFNoCorner – cannot be placed in a corner
        - 3.3.2.5.13.7 kWFCornerOnly – can only be placed in a corner
    - 3.3.2.5.1 kSlotID – the index of the slot that this object is contained in. Only supposed to be set by the object moving code.
    - 3.3.2.5.2 kFamilyNumber – the family number of this object. 0 means no family, visitors.
    - 3.3.2.5.3 kCounter1 – generic counter. Not currently used.
    - 3.3.2.5.4 kTrapCount – state variable owned by primitive goto relative. Shares the same address as kCounter1. If 0, the person is not currently involved in routing. If greater than 0, it is the number of tries the person has left to get the route right. If less than 0, it is the object id of the portal object that failed. This is so cascaded routes can tell when they have failed.
    - 3.3.2.5.5 kRoomCompDelay – doors only. Shares same address as kCounter1. Not currently used. Formerly it was decremented each tick by cXDoor::Simulate, and when it reaches 0, the tree table of the door was recomputed to reflect all the objects in the opposite room.
    - 3.3.2.5.6 kAge – age of the object. Currently not used explicitly.
    - 3.3.2.5.7 kGender – gender of the object. Only used by behavior trees.
    - 3.3.2.5.8 kTreeTableEntry – current tree table interaction being examined. Set by certain primitives like “Find Action New” (and “Expression”, which can set any data field), and used by Primitive “Gosub Found Action” to invoke the interaction it refers to.
    - 3.3.2.5.9 kSearchRadius – currently not used. Foremerly used to denote how far the object could “see” when searching for other objects.

3.3.2.5.10 kSpeed* – how fast a person walks in fixed point frac tile increments.
3.3.2.5.11 kRotationSpeed – how much to increment or decrement the object's direction when rotating.
3.3.2.5.12 kCounter2 – not used
3.3.2.5.13 kRouteCount – shares address space with kCounter2. Not currently used. Foremerly a step counter for routing.
3.3.2.5.14 kUseCount – timer for when an object will be made available for interactions again. Currently is being set by behavior trees and decremented in the Simulate routine, but is not being checked by the "Find Action New" primitive.
3.3.2.5.15 kContainerID – object id of the object that contains this. Supposed to only be set by the object moving code.
3.3.2.5.16 kWeight – representation of the object's weight. Use to determine if an object is too heavy to place on another object.
3.3.2.5.17 kSupportWeight – how much weight the object can contain. Used to determine if another object can be placed on this one.
3.3.2.5.18 kRoom – the room of the object.
3.3.2.5.19 kRoomPlacement – value denoting where an object can be placed with respect to rooms. Takes on one of kRmPlAnyRoom, kRmPlOutsideOnly, kRmPlInsideOnly.
3.3.2.5.20 kHidden – whether or not the object is hidden.

3.3.2.6 SInt16 fAttrs[kNumAttr] – Attribute Array. Currently kNumAttr is 8. Accessed directly by behavior trees through expression primitive, but not at all by compiled code. Used within object types to store state information that does not need to interact with compiled code.

3.3.2.7 SInt16 fTemp[kNumTemp] – Temp Array. Accessed directly by behavior trees using expression primitive. Used to pass values around, but not guaranteed to keep its value across a sim tick.

3.3.2.8 SInt16 fIterations – this is duplicated from TreeSim. This is an error.

3.3.2.9 SInt16 fDirInc – not used anymore. Used to be a seed value for route planning.

3.3.2.10 RTSlotList fSlots – The list of slots for this object. Some slot values change as the part of the state. That is why slots are mentioned in properties and state.

3.3.2.11 Boolean fIsDirty – indicates if the object need to be drawn.

3.3.2.12 Int fHiliteMask – a mask indicating the current hiliting of the object. Passed down to the blitters and vitaboy to render objects in different hilite states. Currently the hilite flags are kHilited, kSelected, kCanInterrupt. The selected person and only the selected person has the kSelected flag set. The tools and the mouse object set the kHilited flag to show an object is being placed. TBD: other hilite modes?

3.4 Object Placement Objects can be located in a number of ways. Starting with objects in the grid, there is an n-ary tree of containment, where the n is limited by the number of slots in an object. The root node is an abstract node with all grid objects as its leaves. Other nodes are objects and can have an object id in each of its statically defined slots. The position of an object in this tree is determined by its data fields, kContainerID and kSlotID, and its location.

3.4.1 Slots. Each slot has a statically defined 3D coordinate in object model coordinates (see coordinates, chapter 2). This is used only when rendering. Each slot has a type field that is one of the following: kContainedObject, kHeadline, kHandle.

3.4.1.1 Headline slots. The following fields are used:

1. SInt16 headlineSpriteID – the sprite id of the headline to draw here.
2. Boolean headlineGlobal – whether or not the sprite id is a global sprite or a local.
3. SInt16 headlineFrame – which frame of the headline is being drawn
4. SInt16 ticksLeft – how many more ticks to draw it for, decremented each tick.

3.4.1.2 Contained Object Slots. An objectID field is used to hold the id of the contained object. This object's location, container id and slot num must match up as described below.

3.4.1.3 Handle Slots. This is the slot which governs the relative origin of a contained object to that of its contained slot. All objects have an impled handle at 0,0,0, which is used if no handles are available. An object can have several handle slots. These determine how the object can be held. This is currently not used in any object, but some code exists

3.4.1 Locales

3.4.1.1 Out Of World This is the state achieved by the polymorhic routine, Pickup. An out of world object has an invalid location and is contained in no other objects (fData[kContainerID] = 0). Pickup has no effect if the object is already out of world.

3.4.1.2 Directly on Grid The object is in the grid: it's id is in the world object array at its tile location. (see World: Chapter 2). The container id of the object is 0. The object may, in this locale, have other objects contained in it. Objects that can be walked over and multi-tile objects must lie directly in the grid (or out of world).

3.4.1.3 Heirarchically contained The object's id is in a containment slot of some other object, whose id is in the container id field. The location is exactly equal to the containing object's, except when the container has the "can walk over" attribute, in which case the location can be anywhere on the tile.

3.4.4 Criteria for placement directly on the grid For a given location, placement of an object there is subject to the following conditions:

1. If the location is out of bounds, the object cannot be placed.
2. If the object has non-zero wall placement flags and the desired location is not in the exact center of a tile, the object cannot be placed.
3. If the wall configuration at the desired tile does not match up with the object's wall placement flags, the object cannot be placed (see 3.3.2.5.13, kWallPlacementFlags).
4. If the room of the tile does not match the room placement flags, the object cannot be placed (see 3.3.2.5.19, kRoomPlacement).
5. If the object can be walked over, it can be placed only if no other object is at that location, or if the object there <u>cannot</u> be walked over. In the latter case, the object is placed under the other object.
6. If the object cannot be walked over, and another object is in the grid that cannot be walked over and is not a part of the given object, the object cannot be placed.
7. If the object cannot be walked over, and another object is in the grid that can be walked over, the object cannot be placed if it cannot be placed <u>on top</u> of the other object (see next section).
8. If the rectangle of any object in one of the 8 adjacent tiles overlaps the would-be rectangle of this object, and the other object is not part of the given object, the object cannot be placed.
9. If the object's would-be rectangle intersects with a wall, it cannot be placed.

3.4.5 Criteria for placement on top of another object. For a given bottom object and slot number, an object can be placed there subject to the following conditions:

1. The slot with that number exists.
2. If the bottom object cannot be walked over and the weight of the object exceeds the support weight of the bottom object, the object cannot be placed.
3. If the object can be walked over, and the bottom object is not part of the object, it cannot be placed.
4. If no object is in the slot or the object there is a part of the given object, the object can be placed. Otherwise it cannot be placed.

Placing

Once an object is picked up, it can be placed directly on the grid or as a contained object in another object. This placement is handled by polymorphic methods CanPlace and Place, both of which take a location in fixed world fractional tile coords and a pointer to an object to place on top of, and a slot number to place in on that object. Both routines in general only go one level deep. That is, if an object cannot be placed at the specified position, it will not attempt to navigate the containment heirarchy.

CanPlace returns a Boolean value, and Place should only be called if CanPlace returns true. CanPlace will only recurse if the object already in the grid has the "can walk over" flag set.

3.2.10 STR# (or cst) : Body Strings

Currently only for people. When a person is instantiated, the body strings are passed to "XVitaboy" and "XAnimator" classes to initialize the bodies and faces of the person. The strings in the table are:

kSkeletonName	skeleton for the person's vitaboy
kSuit1Name	main suit
kSuit2Name	secondary suit (accessories)
kHeadName	suit that is the head
kHeadRegName	name of head's reg. point (currently is "HEAD" for all people)
kRightHandName	name of right hand bone
kRightHandRegName	name of right hand reg. point
kDefaultFaceName	normal face suit
kHappyFaceName	happy face suit
kSadFaceName	sad face suit
kMadFaceName	mad face suit
kSleepFaceName	sleepy face suit

TBD: more fields will be needed, what are they? TBD: Standard accessories?