Technote 1130

Introducing PuppetTime™: Adding New Media Types to QuickTime

By deeje cooley
Apple Worldwide Developer Technical Support

CONTENTS

This Technote describes how to add new media types to QuickTime, and uses the concept of digital actors as an example.

3D media is very exciting to use, but it is beyond the ability of nmost users to create. I definitely believe that 3D is the medium of the future, yet I am constantly frustrated by the learning curve associated with high-end 3D modeling and animation tools. For professionals, these tools are the cream of the crop; however, I want something a little more progressive. I want to manipulate 3D objects that know how to animate themselves, and can interact with each other. Drag a dog onto a stage, then tell it to wag its tail. Drag a cat onto a stage, then tell it to walk around. Put the two together, and tell the dog to chase the cat. In short, I don't want to make 3D objects; I want to use 3D objects to create other things. That's why I've created the PuppetTime™ architecture.

Defining PuppetTime

PuppetTime is an open architecture for digital actors, built on top of QuickTime. What, you may ask, is a digital actor? You may have heard the term before. In its most basic definition, a digital actor is a graphic representation on the computer screen that can accept messages to animate itself. I use the term puppet to mean digital actor, because I want to emphasize the metaphor of virtual strings controlling a shape's appearance and implied behavior.

Here's an example: on my computer screen I have a humanoid-shaped puppet and a list of commands. When I clickon a command, it is sent to the puppet, which then responds with some kind of activity. If I click on "walk" and thenclick on a new location on the screen, the puppet will"walk" to that new location. If I click "wave", the puppet will wave to say "hello." Different puppets might animate themselves in different ways to represent "walk" or "wave"; the power lies in the fact that "walk" and "wave" are now abstracted out, and any

number of puppets can understand these commands while presenting a unique visual appearance for each.

There are many companies now developing digital actors for use in movies and games, but as of yet there is no proposed standard framework which might make the commercial acceptance and distribution of digital actors feasible. The PuppetTime architecture attempts to address this need.

PuppetTime uses the Component Manager and `QTAtoms`, defines a new component interface called puppets, and includes both a derived media handler and a movie import component. The PuppetTime framework is designed with a philosophy similar to QuickTime: it contains a set of toolbox routines for manipulating the PuppetTime media data, as well as a number of extensible components and component interfaces. Much like Sprites and QuickTime Music Architecture in QuickTime, PuppetTime can be used by itself in your applications, and it can also be contained in QuickTime movies alongside other media types like music, text, sound, and video.

You are highly encouraged to have a copy of the PuppetTime Sample Code on hand while reading this document, since I'll refer to its contents often. You can obtain sample code from PuppetTime.com. A royalty-free license is available for the PuppetTime runtime, and third-party and co-development is highly encouraged.

Basic PuppetTime Architecture

The PuppetTime architecture is implemented using several QTML technologies, and defines three key elements: *puppets* , *events* , and the *conductor* .

Puppets

Puppet components are defined and implemented using the Component Manager 3.0 and display themselves using QuickDraw 3D 1.5.x. The puppet component interface and several of the built-in puppets are discussed in detail below.

Events

A PuppetTime event is a `QTAtom` data structure that contains information about a command or action that a puppet should perform. When a puppet receives an event, it pulls out the relevant information and (often) performs some form of animation. A stream of events can come from numerous sources, such as a network connection, or a QuickTime movie track. The standard PuppetTime event format, toolbox routines, and some basic event vocabularies are discussed below.

Conductor

The PuppetTime conductor component acts as the gluebetween events and puppets. At its basic level, the conductor creates the QuickDraw 3D environment, instantiates a number of puppets into the environment, and then receives a stream of events from an external source and re-directs them to the individual puppets. Again, the PuppetTime conductor is discussed below.

Goals of PuppetTime

There are several key design goals for PuppetTime to ensure its acceptability and future growth:

Open Architecture

PuppetTime is designed to be an open architecture. The format for PuppetTime events structures allows for a variable number of bits of information, so that new events can be defined and existing events augmented. The puppet component interface allows new puppets to be added seamlessly to existing PuppetTime-savvy applications.

Compact Data Format

The PuppetTime event format is crafted to allow for the widest range of event vocabularies possible, while keeping an eye towards compactness. PuppetTime uses events as meta-data to recreate a scene at runtime, and as such, events are much smaller than pre-rendered, compressed image samples.

Internet-ready

PuppetTime is designed with web- and internet-savvy applications in mind. For example, on-line 3D comic strips would be possible by downloading a set of puppets once, then delivering new episodes on web pages as QuickTime movies. Each episode movie can be significantly smaller than a pre-rendered 3D scene because it only contains a sequence of events. The puppets themselves can be designed and implemented such that they can update themselves with new capabilities on the fly.

Scalable Performance

PuppetTime performance scales with improvements in CPU speed and internet bandwidth. Because puppets animate themselves in real-time, their visual displays can improve with faster computer systems. In addition, faster Internet connections allow for richer, higher-fidelity event streams.

QuickTime Integration

PuppetTime is designed to work as a new media type within QuickTime. To start, PuppetTime includes a MIDI file importer and a media handler, which allows PuppetTime event streams to be stored and played back within a QuickTime movie, alongside other media types (e.g., music and text).

Technology Map

Figure 1 shows the basic QuickTime architecture with the integrated PuppetTime media type and its related components.

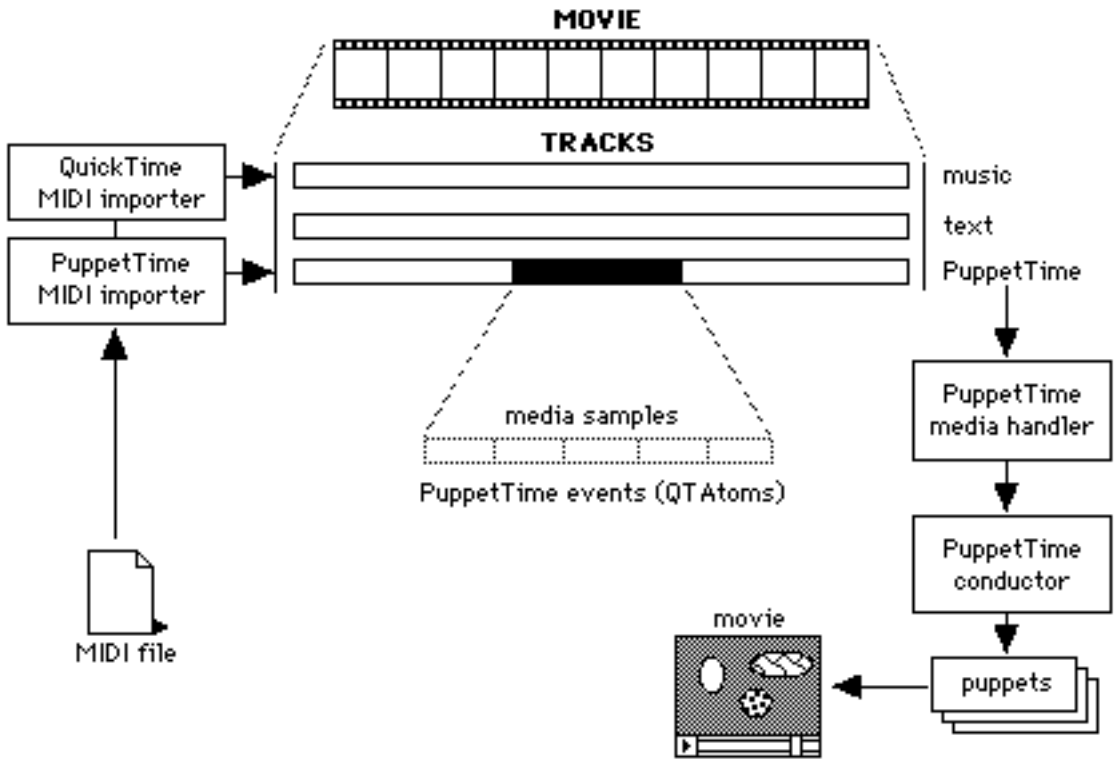

Figure 1. PuppetTime inside QuickTime.

Events

Let's begin our detailed discussion of PuppetTime with events. As mentioned above, a PuppetTime event is a `QTAtom` structure which contains bits of information that describes an action or command to a puppet.

`QTAtoms`

`QTAtoms` are structures that store a variable number of name-data pairs. `QTAtoms` are similar in concept to AppleEvent records, except that `QTAtoms` do not store data-type information, and the API is available for all platforms that QuickTime supports. The QuickTime 3.0 developers guide describes `QTAtoms` in detail, and can be found at <http://quicktime.apple.com/>. As shown in Figure 2, `QTAtoms` can be nested inside one another to create hierarchical data structures.

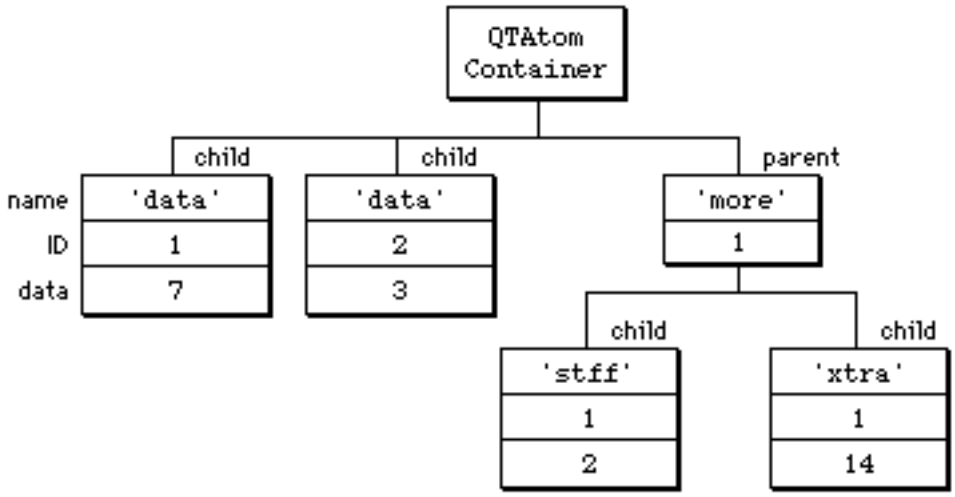

Figure 2. A typical `QTAtom` structure

Listing 1 shows how to create the `QTAtom` structure shown in Figure 2. Note that proper error checking is not included in the listings below, and you should always check for programmatic and runtime errors in your code.

```
QTAtomContainer       aContainer = nil;
QTAtom                anAtom = nil;
long                  aLong = 0;

    // create the QTAtomContainer
anError = QTNewAtomContainer(&aContainer);

    // add some name-data pairs to the root
aLong = 7;
anError = QTInsertChild(
            aContainer,     // the container
            0,              // the atom, zero = root
            'data',         // the name
            1,              // the ID
            0,              // the index of name-ID pairs
            sizeof(aLong),  // the size of the data
            &aLong,         // the pointer to the data
            nil);           // returns a ref to the new QTAtom

aLong = 3;
anError = QTInsertChild(aContainer,
                        0,
                        'data',
                        2,
                        0,
                        sizeof(aLong),
                        &aLong,
                        nil);

    // create an empty atom
anError = QTInsertChild(aContainer,
```

```
                            0,
                            'more',
                            1,
                            0,
                            0,
                            nil,
                            &anAtom);

    // add some atoms to it
aLong = 2;
anError = QTInsertChild(aContainer,
                            anAtom,
                            'stff',
                            1,
                            0,
                            sizeof(aLong),
                            &aLong,
                            nil);

aLong = 14;
anError = QTInsertChild(aContainer,
                            anAtom,
                            'xtra',
                            1,
                            0,
                            sizeof(aLong),
                            &aLong,
                            nil);

    // make sure to dispose of it when you're done
anError = QTDisposeAtomContainer(aContainer);
```

Listing 1

Basic Structure

Thus, a PuppetTime event is a `QTAtom` structure with a well-defined set of name-data hierarchy, shown in Figure 3 (`QTAtom` IDs are not used by the PuppetTime toolbox routines, and will be omitted from the following figures).

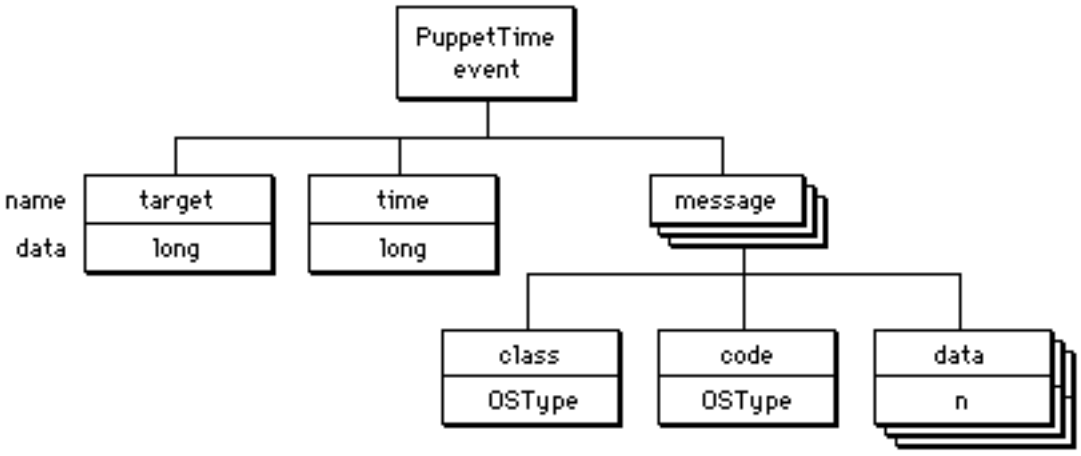

Figure 3. The PuppetTime event structure

A PuppetTime event includes the following atoms:

- Target - this atom contains an ID of the target puppet for this event. Puppet IDs are assigned to puppets at runtime, so any puppet can be used, and the event always goes to the puppet with the given ID.
- Time - this atom contains the time that this event should occur. This time value will be relative either to the start time of a movie or to the system clock. The puppet is responsible for queuing this event until the given time has arrived. A queuing method is provided to any puppet that wants it (explained below).
- Messages - each event contains one or more messages. Each message contains a message class/code combination and zero or more parameters. In this way, a number of messages can be sent to a puppet with only one event.
- Class - contains the message class being invoked (e.g., 'core' or 'musi').
- Code - contains the message code, (e.g., 'walk' or 'wave').
- Data - each message can contain a number of parameters, as defined by the creator of the event suite. The parameters can augment the resulting behavior and/or animation (e.g., speed of walk, or exaggeration of wave).

There are constants defined in the header file `PuppetTimeEvents.h` for the event and message names used to build a PuppetTime event, to ensure that all events have the same structure.

PuppetTime Toolbox Routines

As noted above, you can use QuickTime toolbox routines to build `QTAtoms` as PuppetTime events, as long as you structure the `QTAtoms` in the basic format described. Because the format is so specific, there are a number of PuppetTime toolbox routines that make creating and parsing PuppetTime events easy. Look at the file `PuppetTimeEvents.h` for a complete list of available APIs.

Music Events

As an example, let's describe a class of events that represent music. Figure 4 shows a typical music event structure:

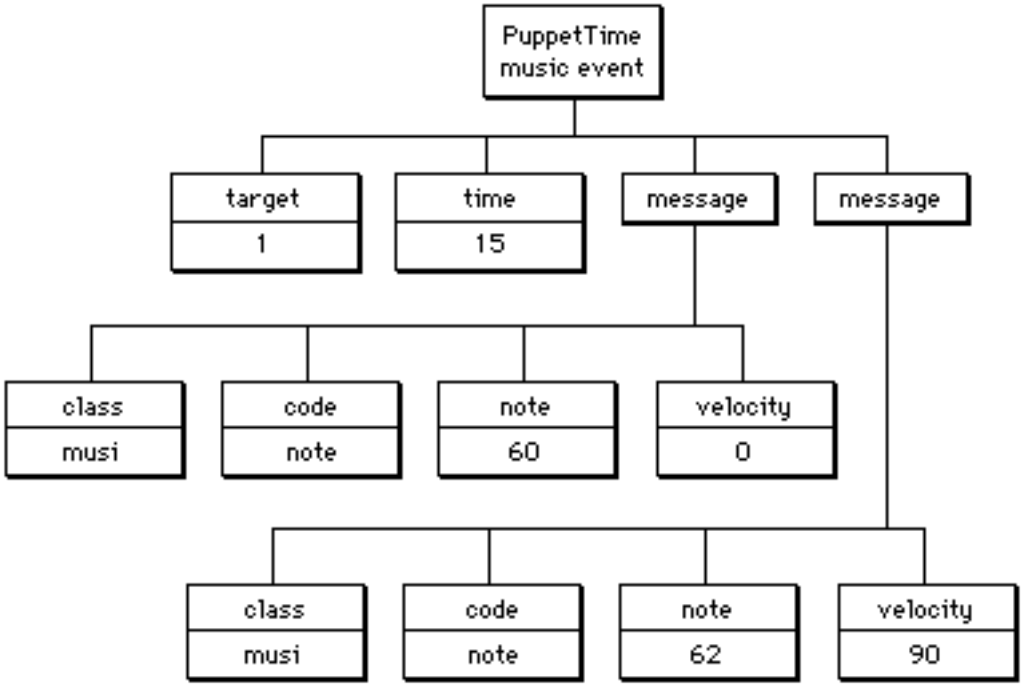

Figure 4. A typical music event

Listing 2 shows how to use the PuppetTime toolbox routines to build the event shown in Figure 4. Notice the PuppetTime toolbox streamlines the hierarchical nature of the event for you.

```
QTAtomContainer MakeAMusicEvent()
{
    SInt32          instrument = 1;
    SInt32          eventTime = 15;
    UInt16          noteNumber;
    UInt16          noteVelocity;
    QTAtomContainer anEvent = nil;
    QTAtomContainer aMessage = nil;
    OSStatus          anError = noErr;

        // create a new message
    aMessage = PTNewMessage(kPTInstrumentClass, kPTNoteEvent);

        // insert the parameters
    noteNumber = 60;
    noteVelocity = 0
    anError = PTSetProperty(aMessage,
                            kNoteNumber,
                            sizeof(noteNumber),
                            &noteNumber);
    anError = PTSetProperty(aMessage,
                            kNoteVelocity,
                            sizeof(noteVelocity),
                            &noteVelocity);

        // create the event
    anEvent = PTNewEvent(instrument, eventTime, aMessage);

        // add the second message
    noteNumber = 62;
    noteVelocity = 90;
    anError = PTSetProperty(aMessage,
                            kNoteNumber,
                            sizeof(noteNumber),
                            &noteNumber);
    anError = PTSetProperty(aMessage,
                            kNoteVelocity,
                            sizeof(noteVelocity),
                            &noteVelocity);

    anError = PTSetNthMessage(anEvent, 0, aMessage);
    anError = PTReleaseMessage(aMessage);

        // do something with the event, like add it to a track

    anError = PTReleaseEvent(anEvent);

    return anEvent;
}
```

Listing 2

Optimizations

To minimize the size of PuppetTime event streams, there are a number of optimizations that can be made when storing or transmitting events.

The first optimization is the concept of default values. When an event is defined by an author in a header file, certain parameters will be defined to have default values. When a developer creates an event, she can omit certain parameters to save space. When the recipient of an event goes to read those parameters and finds none, she can assume a default value. So for the music event example above, the default value for the velocity is zero, and that "note off" messages can contain one less parameter. The savings can add up quickly in a PuppetTime track that represents a song visually. Default value optimizations should be done by the creator of the events.

The second optimization is the concept of event flattening. Oftentimes an event will contain only one message, so the contents of the message atom (class, code, and parameters) are moved into the event atom, and the message atom is removed. The PuppetTime toolbox routine `PTOptimizeEventList` performs event flattening.

Puppets

Now we turn our attention to PuppetTime puppets. PuppetTime defines a puppet component interface in the file `PuppetTimeComponents.h`, and also includes a number of puppet components that are used throughout the PuppetTime environment.

The Puppet Component Interface

Listing 3 shows the component interface for puppet components:

```
pascal ComponentResult PuppetInitialize
                        (PuppetComponent puppet,
                    ConductorComponent aConductor);

pascal ComponentResult PuppetSetTimeFormat
                        (PuppetComponent puppet,
                    UInt32 eventTimeFormat);

pascal ComponentResult PuppetIdle
                        (PuppetComponent puppet,
                         UInt32 atMediaTime);

            // message routines
pascal ComponentResult PuppetProcessActionEvent
                        (PuppetComponent puppet,
                         QTAtomContainer anEvent);

pascal ComponentResult PuppetProcessMessage
                        (PuppetComponent puppet,
                         UInt32 atMediaTime,
                         QTAtomContainer aMessage);

            // QD3D routines
pascal ComponentResult PuppetSubmit
                        (PuppetComponent puppet,
                         TQ3ViewObject theView);

pascal ComponentResult PuppetGetGroupObject
                        (PuppetComponent puppet,
                         TQ3GroupObject* aGroup);
```

```
pascal ComponentResult PuppetGetTranslateObject
                        (PuppetComponent puppet,
                         TQ3TransformObject* aTransform);

pascal ComponentResult PuppetGetCameraObject
                        (PuppetComponent puppet,
                         Rect* graphicsBox,
                         TQ3CameraObject* aCamera);
```

Listing 3

The key routines are described below.

- `PuppetInitialize` - this routine is called when an instance of the puppet is created. The puppet should initialize its internal structures, which usually includes creating some geometries in QuickDraw 3D that serve as the puppet's basic visual representation.
- `PuppetProcessActionEvent` - this routine is called when an event is dispatched to a puppet. The puppet should check the time of the event, and queue the event if the current time is less than the event time.
- `PuppetProcessMessage` - when a puppet decides to execute an event, it should send all messages to this routine. This abstraction between processing an event and a message will become clear below when we talk about the base puppet component.
- `PuppetIdle` - this routine is called repeatedly while the puppet is active, and the puppet should do whatever processing is necessary. Often times, in response to an event, the puppet will animate itself, and this is the routine that should handle the next "frame" of the animation sequence. It is up to the puppet to decide how to implement its animation. Most puppets perform their animations by adding, changing, and removing geometries in the QD3D environment.
- `PuppetSubmit` - this routine is called for each puppet when the 3D environment is being drawn. The puppet is responsible for submitting its QuickDraw 3D geometries.

Base Puppet

As you can see, there are a number of routines in a puppet component, and every puppet should implement all of them. But most puppets need the same internal organizations, like a queue for events that aren't quite ready for processing. Moreover, processing events and pulling out messages is the same for most puppets.

To make the process of creating a new puppet for PuppetTime easier, most developers can create a derived puppet component. A derived puppet uses the services of a base puppet component as a delegate to its own code. Similar in concept to a base media handler or a base image decompressor, the base puppet component implements the basics of a puppet, and leaves the specifics of the geometries and animations to the developer.

To create a derived puppet component, a developer must implement the following puppet component routines: `PuppetOpen`, `PuppetClose`, `PuppetInitialize`, `PuppetIdle`, and `PuppetProcessMessage`.

The `PuppetOpen` routine should make an instance of the base puppet component and set derived puppet component to be the target. Listing 4 shows a simple derived `PuppetOpen` routine:

```
pascal ComponentResult PTBlockyPuppetOpen(ComponentInstance self)
{
    ComponentResult     result = noErr;
    PTBLPrivateGlobals**    storage = NULL;

    storage = (PTBLPrivateGlobals**)
                        NewHandleClear(sizeof(PTBLPrivateGlobals));
    if (storage != NULL)
    {
            // store our globals in the component instance
        SetComponentInstanceStorage(self, (Handle) storage);
        (**storage).self = self;

            // get the Blocky media handler component
        (**storage).delegate =
                            OpenDefaultComponent(PuppetComponentType,
                                                 BasePuppetComponentType);
        ComponentSetTarget((**storage).delegate, self);

            // initially we target ourselves
        (**storage).target = self;
    }

    return (result);
}
```

Listing 4

In `PuppetClose`, make sure to release your instance of the base puppet.

In `PuppetInitialize`, your puppet component must call through to the base component, and then create some geometries. Listing 5 shows a simple derived `PuppetInitialize` routine. Notice that the base puppet creates the QD3D group object, and that the derived puppet asks for it using the puppet component routine `PuppetGetGroupObject`.

```
pascal ComponentResult
PTBlockyPuppetInitialize(PTBLPrivateGlobals** storage,
                ComponentInstance aConductor)
{
    TQ3GroupObject      aGroup = nil;
    TQ3GeometryObject     myBox;
    TQ3BoxData          myBoxData;
    TQ3SetObject          faces[6];
    short               face;
    TQ3ColorRGB        faceColor;
    TQ3ColorRGB        faceSeeThru;
    ComponentResult     anError;

    anError = PuppetInitialize((**storage).delegate, aConductor);
    anError = PuppetGetGroupObject((**storage).target, &aGroup);

        // set up the colored faces for the box data
    myBoxData.faceAttributeSet = faces;
    myBoxData.boxAttributeSet = nil;
        // set up some color information
    faceColor.r = faceColor.g = faceColor.b = 0.8;
    faceSeeThru.r = kNoteTransparency;
    faceSeeThru.g = kNoteTransparency;
    faceSeeThru.b = kNoteTransparency;
    for (face = 0; face < 6; face++)
    {
        myBoxData.faceAttributeSet[face] = Q3AttributeSet_New();
        ::Q3AttributeSet_Add(myBoxData.faceAttributeSet[face],
                        kQ3AttributeTypeDiffuseColor,
                        &faceColor);
        ::Q3AttributeSet_Add(myBoxData.faceAttributeSet[face],
                        kQ3AttributeTypeTransparencyColor,
                        &faceSeeThru);
    }
        // set up te basic properties of the box
    ::Q3Point3D_Set(&myBoxData.origin, 0, -(6 * kNoteSize), 0);
    ::Q3Vector3D_Set(&myBoxData.orientation, 0, 12 * kNoteSize, 0);
    ::Q3Vector3D_Set(&myBoxData.majorAxis, 0, 0, kNoteLength);
    ::Q3Vector3D_Set(&myBoxData.minorAxis, kNoteWidth, 0, 0);

        // create the box itself
    myBox = ::Q3Box_New(&myBoxData);
    ::Q3Group_AddObject(aGroup, myBox);
    ::Q3Object_Dispose(myBox);

        // dispose of the objects we created here
    for( face = 0; face < 6; face++)
    {
        if (myBoxData.faceAttributeSet[face] != nil)
            ::Q3Object_Dispose(myBoxData.faceAttributeSet[face]);
    }

    return anError;
}
```

Listing 5

In the routine `PuppetIdle`, you can do whatever idle time processing you like. Just make sure to give the base puppet some idle time as well. Listing 6 shows how:

```
pascal ComponentResult
PTBlockyPuppetIdle(PTBLPrivateGlobals** storage,
            UInt32 atMediaTime)
{
    ComponentResult     anError;
    anError = PuppetIdle((**storage).delegate, atMediaTime);
    for (short i = 0; i < kNumberOfNotes; i++)
    {
        if ((**storage).fNotes[i] != nil)
        {
           anError = (**storage).fNotes[i]->Idle(atMediaTime);
        }
    }
    return anError;
}
```

Listing 6

When the base puppet decides to pull an event from its queue, it reads each of the messages inside it and sends them to `PuppetProcessMessage`. This is where your puppet can receive its messages and perform its animations. Notice that the switch statement defaults to calling back into the base puppet, which can handle certain basic messages on its own (e.g., "locate at"). Listing 7 shows an example:

```
pascal ComponentResult
PTBlockyPuppetProcessMessage(PTBLPrivateGlobals** storage,
                    UInt32 atMediaTime,
                    QTAtomContainer aMessage)
{
    ComponentResult anError = noErr;
    OSType          messageCode;

    ::PTGetMessageCode(aMessage, &messageCode);
    switch (messageCode)
    {
        case kPTNoteEvent:
        {
            anError = ProcessNoteMessage(storage, aMessage);
            break;
        }

        default:
            anError = PuppetProcessMessage((**storage).delegate,
                                atMediaTime,
                                aMessage);
            break;
    }

    return anError;
}
```

Listing 7

As you can see, a base puppet handles a lot of th details for you, allowing you to concentrate on implementing your puppet's visual appearance and animations. Note also the object-oriented nature (i.e. inheritance and overriding) of using a base puppet inside your puppet. The Component Manager was designed specifically for this kind of use.

Camera Puppet

Another important puppet in PuppetTime is the camera puppet. This is a derived puppet which provides a view of the PuppetTime world to the user. Any puppet can have a camera view associated with it, although it's not required. The camera puppet is special in that it has no geometry associated with it, and simply provides a view.

The file `PuppetTimeCameraEvents.h` defines a vocabulary for camera control, and the file `PuppetTimeEvents.h` includes the core vocabulary for basic movement. This means that the view can be changed by sending "move" events to the camera puppet. Just like all other PuppetTime events, these "move" events can be generated at runtime based on user input devices (e.g., a joystick), or can be stored along with otherevents in an event stream (e.g., panning during playback of a movie). At this time, there is only one camera puppet allowed: future versions of PuppetTime will expand the role and use of camera-enabled puppets.

Creating your own puppets

There are several puppets with sample code available in the SDK, which demonstrate the proper way to use the base puppet component. Use these example projects as the basis for your puppet development.

Make sure that you edit the 'thng' resource, and choose mixed- or upper-case constants for the subtype

and manufacturer fields. At this time, there are no flags defined for puppets, so zero them out for now.

When implementing puppets, you'll have to decide what vocabularies to support. There are several sets of vocabularies already defined in PuppetTime, such as core and music. The base puppet handles a number of the core events for you.

You're also free to create your own vocabularies, but you should use your manufacturer code as the message class; you'll also have to generate PuppetTime tracks using your vocabularies.

Music Puppets

For music puppets, the file `PuppetTimeMusicEvents.h` contains all the constants for the music vocabulary. The file `PuppetTimeMusicEvents.c` includes several utility routines to easily build music events.

The PuppetTime MIDI import component, discussed below, creates PuppetTime tracks using the music vocabulary. This allows a user to quickly generate PuppetTime content by simply importing MIDI files into QuickTime movies using applications like MoviePlayer.

Conductor

Next we examine the heart of PuppetTime, the conductor. It is the central object that binds the puppets to the drawing environment and to the incoming event stream.

3D Environment

When an instance of the conductor component is created, it in turn instantiates a number of QuickDraw 3D objects to set up a drawing environment, including a renderer, viewer, context, etc.

Each puppet is responsible for its own geometries, yet this information needs to be communicated to the conductor at some point. This is done when the conductor is instructed to draw: each puppet gets a chance to submit its geometries (and other objects) to the QuickDraw 3D rendering loop maintained by the conductor.

Event Dispatching

Besides being responsible for the overall display, the conductor is also responsible for dispatching events from an incoming events stream to the puppet instances.

There is a class of events specific to the conductor, like the 'cast' event. Events targeted to the conductor have a target ID of zero. A cast event has the structure shown in Figure 5:

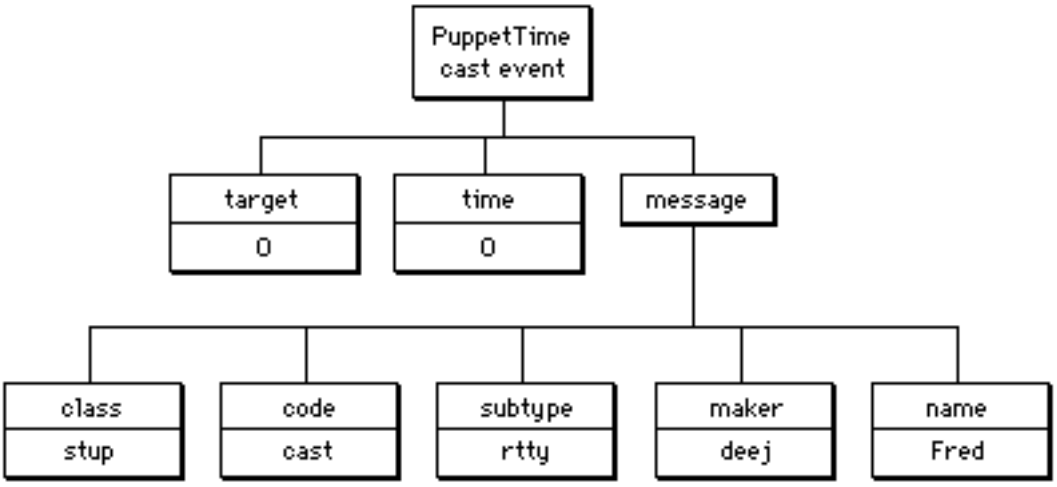

Figure 5. A cast event structure

The cast message contains the following fields:

- `kPuppetSubType` - this field contains the subtype of the puppet component to instantiate, or cast. This field corresponds to the subtype field in the 'thng' resource that defines your puppets. When casting a puppet, the conductor will search for a component where the type is 'PTpt' and the subtype is the value in this field.
- `kPuppetMaker` - this optional field allows you to further identify your puppet component, which matches the manufacturer field in your 'thng' resource. When casting your puppets, you should use this field to avoid name collisions, which can result in the wrong puppet being cast.
- `kPuppetName` - this field is also optional, but if it's present, it will be used in future version of PuppetTime for things like onscreen identification and event targeting via name.

When the conductor receives cast events, it examines the contents of the event to determine which puppet to instantiate, then creates and stores a puppet instance in its internal array.

Cast events are among the first events passed to a conductor. Without them, there would be no puppets visible and nothing to dispatch further events to. You can use the routine `PTAddCastingEventToList` to easily add a cast event to an event list. Note that a cast event doesn't tell the puppet where it should be when it is created; therefore, you should also add a locate event to the event stream using the routine `PTAddLocateEventToList`.

We'll talk about cast events as they pertain to QuickTime movies below.

Current Camera

The conductor has the concept of a current camera, which is itself a puppet. When a conductor first initializes, and after it has created the QuickDraw 3D drawing environment, it casts a camera puppet and assigns it a special ID `kPTDefaultCamera`. This gives the conductor an initial view in which to draw.

Events can be targeted to the default camera by using an ID of `kPTDefaultCamera`. Because the camera object uses the base puppet, it understands many of the core vocabulary, like "move to" and "turn." The first version of PuppetTime supports only one camera, but future versions will expand on this.

QuickTime integration

The first part of this article described how PuppetTime is structured around `QTAtoms` and puppet components. Nov on how PuppetTime is integrated with QuickTime. The initial release of PuppetTime allows for basic creation and p QuickTime movies containing PuppetTime tracks.

PuppetTime Tracks

A PuppetTime track in a QuickTime movie has the type 'PTmh'. Each sample in a PuppetTime track is a `QTAtomCo` structure containing a list of PuppetTime events. The events in this list represents at minimum 2-3 seconds worth o the duration of each sample can be much larger. This ensures that disk access is minimized for better playback perf

Playing PuppetTime Tracks

In QuickTime, each track type has a corresponding media handler component which handles the playback of the tra contents. Thus, a video track is managed by an instance of the video media handler, and a sound track is managed instance of the sound media handler.

The PuppetTime media type is no different: a PuppetTime track is managed by the PuppetTime media handler, whi derived media handler. When QuickTime opens a movie and finds a PuppetTime track, it searches for the correspo PuppetTime media handler (by finding a component of type 'mhlr' and a subtype equal to the track type, in this ca and creates an instance.

Being a derived media handler, many of the functions are handled by the base media handler component. The Pupp media handler does handle certain routines itself, and the most notable are `MediaInitialize` and `MediaIdle`.

`MediaInitialize` function

During movie initialization, QuickTime calls the media handler routine `MediaInitialize`. Here, the PuppetTime m handler creates an instance of the PuppetTime conductor and tells it about the visual dimensions of the track.

Next, it looks for some cast data associated with the track. The cast data is stored separately so that the cast of the m be easily changed. For example, in a PuppetTime track that contains music events, the actual puppets used during p can be changed by modifying the cast data. The resulting visual representation will be different while the underlyin stream remains valid. Use the routines `PTGetCast` and `PTSetCast` to get and set the cast data for the track.

`MediaIdle` function

If the conductor is the heart of PuppetTime, then the `MediaIdle` function is the heartbeat of a PuppetTime track. Th is responsible for reading in samples from the underlying media and passing them off to the conductor for dispatch gives idle time to the conductor so that it can draw.

Creating PuppetTime Tracks

Of course, playing a PuppetTime track is only useful if you have a PuppetTime track. Creating a PuppetTime track QuickTime movie is simple. As explained above, each sample is a list of events; in this way, the samples are space that the disk isn't accessed too often. Listing 8 shows the sample description record, which is rather uncomplicated demonstrates the concepts behind adding a PuppetTime media to a movie.

```
typedef struct PTMHDescription {
    long    size;       // Total size of struct
    long    type;       // kPTMediaType
    long    resvd1;
    short   resvd2;
    short   dataRefIndex;
    long    version;
} PTMHDescription, *PTMHDescriptionPtr, **PTMHDescriptionHandle;
```

Listing 8

```
Track                   myTrack;
Media                   myMedia;
QTAtomContainer         anEventList;
PTMHDescriptionHandle   aDesc = nil;
TimeValue               sampleTime;

myTrack = NewMovieTrack(theMovie,
                                (long) myWidth << 16,
                                (long) myHeight << 16,
                                0);

                                                        // create the media for the
myMedia = NewTrackMedia(myTrack,                        // the track
                                kPTMediaType,           // the type of media
                                aTimeScale,             // time scale
                                nil,                    // data ref
                                (OSType) nil);          // type of data ref

anEventList = PTNewEventList();
anError = PTAddLocateEventToList(anEventList,
                                        1,              // target
                                        0,              // time
                                        0,              // x
                                        0,              // y
                                        0);             // z
anError = PTAddNoteEventToList(anEventList,
                                        1,               // target
                                        5,               // time
                                        60,              // note
                                        95,              // velocity
                                        0);              // duration (0=forever)
anError = PTAddNoteEventToList(anEventList,
                                        1,               // target
                                        65,              // time
                                        60,              // note
                                        0,               // velocity (0=off)
                                        0);              // duration (0=forever)

aDesc = (PTMHDescriptionHandle) NewHandleClear(sizeof(PTMHDescription));
(**aDesc).size = sizeof(PTMHDescription);
(**aDesc).type = kPTMediaType;
(**aDesc).version = kPTMediaVersion;

                                                        // Start editing se
```

```
anError = BeginMediaEdits(theMedia);

                                                        // add the data to
anError = AddMediaSample(theMedia,
                                (Handle) anEventList,   // the sample
                                0L,                     // offset
                                GetHandleSize((Handle) anEventList),
                                65,                     // duration of samp
                                (SampleDescriptionHandle) aDesc,
                                1,                      // number of sample
                                0,                      // sample flags
                                &sampleTime);           // returned time

                                                        // end editing sess
anError = EndMediaEdits(theMedia);

                                                        // append to the tr
anError = InsertMediaIntoTrack(theTrack,                // the track
                                    -1,                 // where to insert
                                    0,                  // where in the med
                                    65,                 // how much media t
                                    1L << 16);          // the media rate
```

Listing 9

Of course, there are routines in the PuppetTime toolbox that make creating PuppetTime tracks even easier, like `PTAddPuppetTimeSample` and `PTSetEventListForTrack`.

PuppetTime movie import component

Users should have an easy way to create PuppetTime tracks from abundant existing content. The initial release of P focuses on music visualization, and includes a movie import component that converts MIDI files into PuppetTime t

QuickTime already includes a movie import component for MIDI files. The trick is to hook into the QuickTime imp component in such a way that while it is creating a music track, PuppetTime gets a chance to create a PuppetTime tr alongside it.

This can be done by capturing the MIDI import component and replacing it with the PuppetTime import componen step beyond just delegating to a component. Capturing means that the PuppetTime import component gets exclusiv MIDI import component, and takes the latter out of the Component Manger's current registry.

In addition, PuppetTime wants to capture the MIDI import component at startup time, so that whenever QuickTime import a MIDI file--regardless of which application is calling QuickTime--the PuppetTime MIDI import component its magic.

Capturing a component at runtime takes a bit of finesse. First, the `thng` resource must be properly configured: the t subtype of the PuppetTime import component must match the component being capturing (in this case 'eat ' and 'M the `cmpWantsRegisterMessage` flag is set to *true* , which tells the Component Manager that the PuppetTime impo component wants its Register routine called at startup. The rest of the component flags should be the same as the co being capturing. Finally, the PuppetTime movie import component is a PPC-native component, so the component `HasMultiplePlatforms` flag is set to *true* . This tells the component manager to find the component in the extend structure.

Now that the component successfully captures the MIDI import component, it needs to override the `MovieExchangeImportFile` function. This routine calls throughto the captured and delegated MIDI import compo

proceeds to create the music track from the MIDI file. After that routine returns, the PuppetTime import component re-reads the MIDI file and creates a PuppetTime track, converting MIDI data structures into PuppetTime events usi music vocabulary. The code could have just as easily read the newly created music track. Either way, without any on the user's part, a new movie is created that contains both a music track and a PuppetTime track.

To make the user experience complete, the PuppetTime import component also overrides the `MovieExchangeDoUs` routine. In this case, however, it doesn't call through to the MIDI import component, but puts up its own Options instead.

Sample Code

In the PuppetTime Sample Code, I've included slightly altered versions of the PuppetTime media handler and the PuppetTime movie import component for your review. You'll note that I've changed all occurrences of the subtype and manufacturer fields to 'XXXX' and 'YYYY'. If you choose to use these samples for the basis of your own projects, please change the constants to something more suitable. Also, please don't re-use my constants for the various PuppetTime components, particularly 'PTmh' for my media handler and media type; this will allow me to continue developing PuppetTime without external complications.

Future Direction

Much like the QuickTime architecture, PuppetTime is designed with future growth squarely in mind.

More QuickTime integration

With the initial release of the PuppetTime engine, only two QuickTime-related components are included: a media handler and a movie import component. As PuppetTime continues to develop and mature, more QuickTime components will be added.

- Sequence grabber channel - A PuppetTime sequence grabber channel will work with the sequence grabber component to capture PuppetTime tracks in real-time from a number of input devices, like keyboards and joysticks. It could also be used to capture a network event stream, such as in a multi-user game environment.
- Movie Controller - PuppetTime is well integrated with other QuickTime media types, but the existing user experience doesn't allow the user/viewer to move around and among the puppets currently being displayed. A PuppetTime movie controller will add a trackpad-like control alongside the other QuickTime movie controller controls that allows the user to move the camera puppet while the movie is playing.
- Movie Info Panel - While not necessarily a formal part of the QuickTime framework, a PuppetTime movie info panel will allow users to change the puppets in the cast for a PuppetTime track.

Cross-platform

Of course, creating a new media type, especially for web and internet applications, isn't as compelling unless it works on both Macintosh and Windows platforms. Near-term future development will focus on bringing the core toolbox and component functionality to both platforms under QuickTime 3.0.

Consumer Applications

Of course, the PuppetTime architecture exists so that developers can create applications that create and edit the PuppetTime media type. A couple of consumer-level applications that I'd like to see happen are the Puppet Builder and the Puppet Scene Maker.

The Puppet Builder application would allow a user to create new puppets, giving them shapes and simple animations, and matching animations to events.

The Puppet Scene Maker would allow a user to create a scene with dialog in 3D. For example, you could drag puppets from a cast window onto a stage window, then enter dialog in the script window. You might also drag actions from a vocabulary window onto the script window to add movement, nuances, etc.

Bibliography and References

Wang, John. "Somewhere in QuickTime: Derived Media Handlers" develop, The Apple Technical Journal, issue 14 (June 1993), pp. 87-92.

Guschwan, Bill. "Somewhere in QuickTime: Dynamic Customization of Components" develop, The Apple Technical Journal, issue 15 (September 1993), pp.84-88.

Inside Macintosh: QuickTime, by Apple Computer, Inc. (Addison-Wesley, 1993).

Inside Macintosh: QuickTime Components, by Apple Computer, Inc. (Addison-Wesley, 1993).

3D Graphics Programming With QuickDraw 3D, by Apple Computer, Inc. (Addison-Wesley, 1995).

Additional Resources:

The QuickTime homepage is at <http://www.apple.com/quicktime/> and the QuickTime developer homepage is at <http://www.apple.com/quicktime/developers/>.

The PuppetTime homepage is at <http://www.puppettime.com/>, where you can download the latest docs, runtime, and SDK.

Downloadables

 Acrobat version of this Note (how many K?)

Acknowledgments

Thanks to Joel Cannon, Scott Kuechle, Gregg Williams, Kathryn Donahue, Steve Cooley, Tony Gentile, and Jason Downs.

To contact us, please use the Contact Us page.
Updated: 25-May-98