

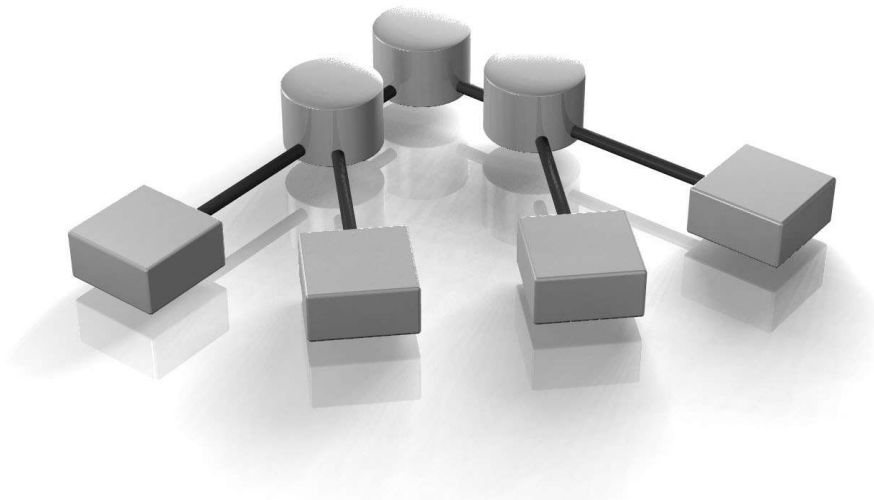
SIXTH FRAMEWORK PROGRAMME
PRIORITY IST-2002- 2.3.1.8
Networked Audiovisual Systems



Uni-Verse Project

WP 3.3 Scripting Environment API
April 29th 2005

Distribution: Public



STREP project

Project acronym: Uni-Verse

Project full title: A Distributed Interactive Audio-Visual Virtual Reality System

Proposal/Contract no.: 002228

Table of Contents

Introduction	3
The Purple Concept	3
Purple Parts.....	3
System View.....	3
Communications	4
Information Publishing	4
Engine Control.....	5
Plug-Ins.....	5
Inputs and Outputs	5
Writing Plug-Ins.....	5
Functional Programming	5
Plug-in Anatomy	6
Built-In Plug-ins.....	6
Graphs	6
The Purple API.....	8
Init-Functions	8
Input- Functions	9
Node- Functions	9
Output- Functions	10
Purple Internals	11
Abstract Data Types.....	11
Node Database	11
Node Synchronization	11
API Implementation	12
Graph Management	12
Application Core.....	12
Conclusions	12
Appendix: Terminology.....	13

Introduction

This document describes the D3.3 deliverable in the Uni-Verse project. The title of this deliverable is “Scripting Environment API”.

Since this is a rather long and unwieldy name, the environment has been codenamed “Purple”, and will be referred to using this name throughout this document.

The Purple Concept

The following is an excerpt from the Uni-Verse specification (D2.3, page 20), about the D3.3 Scripting API:

“The purpose of the Uni-Verse scripting API (codenamed Purple during development) is to design and implement a system through which we can lower the threshold for working with Verse, for both end-users (primarily 3D graphics package users) and programmers.”

In this document, the D3.3 deliverable will be described, with the intent of seeing how well the outlined goals have been reached.

In practice, the Purple concept is implemented as a plug-in based computational engine, that uses Verse as its back-end database and that is fully remotely controllable.

What this means is that Purple creates a way for developers to write small modules that encapsulate some functionality or tool, such as “create a cube model”, “warp this geometry along the Y axis by this many degrees”, “create a bitmap with this pixel data content” and so on. These modules, or plug-ins, are then loaded by the engine and made available for use. End-users control the operations remotely, using the Verse protocol and data model for the communications as needed.

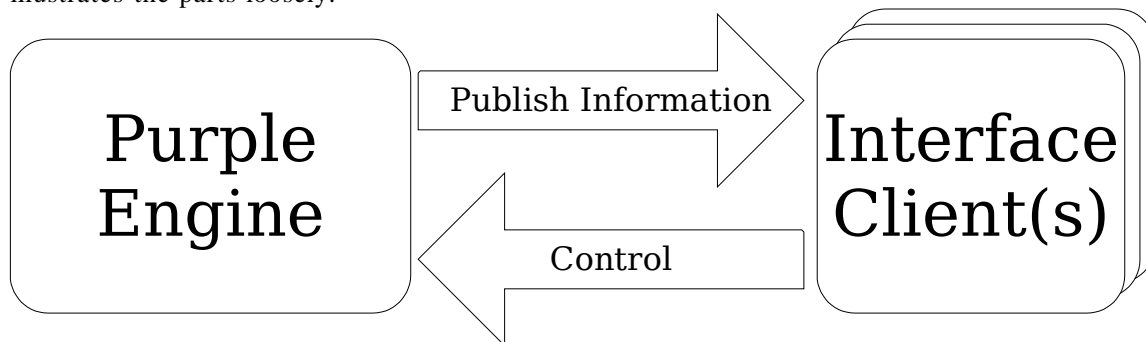
Since an individual plug-in doesn't do much work, Purple supports connecting several plug-ins together into “graphs”. A graph could begin by using a cube plug-in to create a cube, send that geometry to a warp plug-in to perform some warping, and then finally send the data out to the Verse host.

Purple Parts

This section describes the abstract parts that make up the Purple system.

System View

Purple is most easily understood when considered as a system, rather than as a single program. The system consists of several parts, not all of which are addressed by this deliverable. Drawing 1 illustrates the parts loosely.



Drawing 1 Purple System Overview

On the left is the Purple Engine, which is the “back end” of the system. On the right is one or more interface clients, which represent actual end users using the system to do something.

Note that the arrows in Drawing 1 do not denote direct paths of communication, they are instead abstract indicators of how information flows in the system. The Purple engine publishes information about various things that it knows about, and interface clients can use this information to issue controlling actions over the engine. Such control can cause the published information to change.

To be a bit more concrete, the information publishing is implemented through the use of Verse text nodes holding data in XML format. The actual publishing (storage and access control) is done by the

Verse host as always. Control of the Purple system is through the use of methods¹ on the Purple engine's avatar object node.

Further sections will describe these parts in more detail.

Communications

As has already been mentioned, there are two main lines of communications within the Purple system: the publishing of information by the engine, and the control of the engine by users. Let's look closer at these two.

Information Publishing

There are several pieces of information that the engine needs to publish to the world, since any user of Purple will need it:

- What plug-ins are available
- Which graphs have been constructed
- What each constructed graph looks like

All of these needs are addressed by making the engine export textual descriptions of the data in question, into standard Verse text nodes.

The two first are information that is “owned” by the engine, and they are both described in a single text node linked to by the Purple engine. The engine creates this node as it starts up, and names it “PurpleMeta”. It then creates two buffers in it: one for a list of available plug-ins, and one for a list of existing graphs.

Both of these lists are in XML format, which might sound like overkill for what is called mere lists of items. But these lists include a lot of detail, and the easily extended structure of XML fits it very well. For instance, the description of a single plug-in can include its author's name, a brief description of its usage, the originating organization, and a detailed description of each of the plug-in's inputs.

Here's an example of how this XML can look, taken from a Verse server:

```
<plug-in id="10" name="cube">
  <inputs>
    <input type="real32">
      <name>size</name>
      <flag name="required" value="true"/>
      <range>
        <def>10</def>
        <min>0.1</min>
        <max>200</max>
      </range>
    </input>
    <input type="uint32">
      <name>splits</name>
      <flag name="required" value="true"/>
      <range>
        <def>1</def>
        <min>1</min>
        <max>100</max>
      </range>
    </input>
    <input type="boolean">
      <name>uv-map</name>
      <range>
        <def>>false</def>
      </range>
    </input>
  </inputs>
</plug-in>
```

This snippet describes a plug-in called “cube”, that accepts three inputs named “size”, “splits” and “uv-map”, respectively.

The third item, descriptions of actual graph content, is solved by letting a user provide a node name and buffer ID to use, when creating the graph. The Purple engine then uses the specified location, and does not create a new buffer in its own text node.

¹ Verse object nodes support the definition and calling of “methods”, which are named procedure-like entry points that can accept a list of parameter values. See the Verse specification for details.

Engine Control

Since users don't have any direct control over the Purple engine (it does not have a graphical user interface of its own), there needs to be some other way to control it. This need has been fulfilled by using Verse's object method support.

All methods used to control a Purple engine are collected in a group called PurpleControl. At the time of writing (April, 2005) there are 20 such methods, that can be divided into three categories:

- Creating and destroying graphs, two methods
- Creating and destroying modules (plug-in instances), two methods
- Setting module inputs, 16 methods

The Verse method system does not support polymorphism, so there needs to be a unique method for each value type to set an input to.

A client that wishes to do something in Purple, such as create a new graph, issues a call of the proper method in the Purple avatar's object node. This call is received by the Purple engine, which will perform the necessary operations, and update the graph index XML in response. The change to the XML is then mirrored out to all subscribers as usual.

Plug-Ins

The concept of the plug-in is very central to Purple; all actual work done using it is done, ultimately, by plug-ins.

Inputs and Outputs

A plug-in is a small program fragment, that has a number of inputs and a single output for delivering some kind of result. Inputs are used to either provide simple operating parameters (such as a tessellation amount, a rotation angle, a size, and so on) or full Verse-standard node data. A plug-in's output consists of a number of "channels"; one channel for each of a set of primitive types, plus one for node data.

It is up to the plug-in programmer to read values from inputs and write things to the output channels, as desired.

Writing Plug-Ins

Plug-ins are written as stand-alone programs, compiled into shared libraries, that are loaded by the Purple engine as it starts up². The engine will run an initializing function in each plug-in, which registers the plug-in with the engine and describes its interface. Typical information provided by a plug-in during initialization include its name, author and other meta information, the number and expected value type for each of the plug-in's inputs, and so on.

To do these things, and—more importantly—to get actual work done, the plug-in must use some kind of interface to the external world, i.e. to the Purple engine. This is provided through the Purple Applications Programming Interface, or API.

Functional Programming

One interesting aspect about Purple plug-ins is that they should try to approximate functional programming. That is, the result of running a plug-in should be a true function of its input values; there should be no hidden dependencies or side effects.

Purple assumes this to be the case, and will only let a plug-in run when its inputs change, assuming the previous output to be constant in the meantime.

Purple plug-ins are written, as we will see below, in C, which is not a language widely used in functional programming circles. A more typical language³ could probably have been used, but C works and ensures low overhead and ease of access for developers. Snippets of C in pre-compiled form are also easily integrated into a running program thanks to standard dynamic library⁴ infrastructure in all target operating systems. This saves Purple from having to include a language interpreter, and cuts down the size of the project.

² In the future, it might be desirable to add support for loading new plug-ins without restarting the Purple engine. Doing so would add flexibility to the work flow.

³ Most "typical" would probably be some variant of LISP.

⁴ Purple plug-ins compile shared objects, known as "DLLs" in some circles.

Plug-in Anatomy

A Purple plug-in has only a single externally visible symbol: a function called `init()`. This function accepts no parameters, and has no return value. It is called exactly once by Purple, some time after the plug-in code has been read in from disk.

It is the responsibility of the `init()` function to describe the plug-in to the engine, so that the engine can publish that information for users, and also so that the engine knows how to make the plug-in “do it's thing”, i.e. look at its inputs and compute a (new) value for the output.

If the `init()` function does not provide the Purple engine with a satisfactory description, the plug-in cannot be used in a meaningful way. Such plug-ins will be removed from the list of loaded plug-ins, and information about them will not be published at all.

Built-In Plug-ins

There are two special tasks that must be solved in order to get a graph to do some real work: it must be possible to access existing Verse data, and it must be possible to create new such data.

These two tasks are solved by a pair of special, built-in plug-ins, that are always available: `node-input` and `node-output`.

The `node-input` plug-in solves the problem of accessing to Verse data in a graph. It has a single input accepting the name of a node, and will output that node (if found). Thanks to being implemented inside the Purple engine, it can use internal features to change its output whenever the external node changes. This means it becomes possible for a Purple graph to track changes to a node by simply depending on it through the use of a `node-input`.

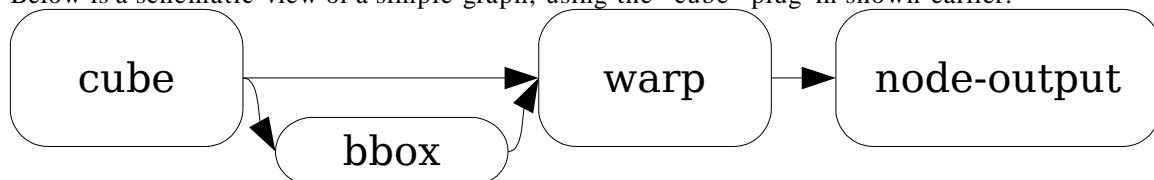
As can perhaps be guessed, the `node-output` plug-in solves the inverse problem: it sends all nodes found on its single input back to the server, by adding them to the internal synchronizer's (see below) work queue.

Graphs

A graph is a collection of plug-in instances, called modules. It is intended (and sometimes assumed) that the modules be connected together to “form” the graph, but strictly speaking that is not necessary. The only way to create a module is to specify which graph is going to contain it, so all modules must always belong to exactly one graph.

Of course, do to something interesting, modules must be connected together so that data can flow between them. Most graphs tend to contain a `node-output` module, to get the results out to the Verse server. It could be argued that doing so should be the default, but making it into a explicit action keeps the design simple.

Below is a schematic view of a simple graph, using the “cube” plug-in shown earlier:



Drawing 2 Sample Graph

This figure shows a graph with four modules, ending in an instance of the `node-output` built-in. From left to right, we have:

1. A `cube` instance that creates a simple cube model (a geometry node with the proper vertex and polygon data, plus an object node linking to that geometry node).
2. Then comes a `bbox` module, which simply computes the bounding box for the first object's geometry node found in the input.
3. Third, an instance of `warp` computes a controllable warp (or twist) along the Yaxis of all geometry inside the bounding box provided.
4. Last, the (now twisted) geometry is sent back to the server, where it can be accessed for viewing by all other connected Verse clients.

Changing any of the input literal values, like the tessellation level for the cube or the warp angle, will cause Purple to re-run the graph starting at which ever plug-in had its inputs changed and propagating as dictated by the connections.

The above figure is slightly simplified, since it doesn't show the individual inputs that many of these plug-ins have, but just the data flow in general. Also, as seen in the XML listing earlier, the `cube`

Uni-Verse Deliverable D 3.3

plug-in has a number of scalar inputs to control its behavior that are also not shown. The following image shows what the results of running the above graph can look like:

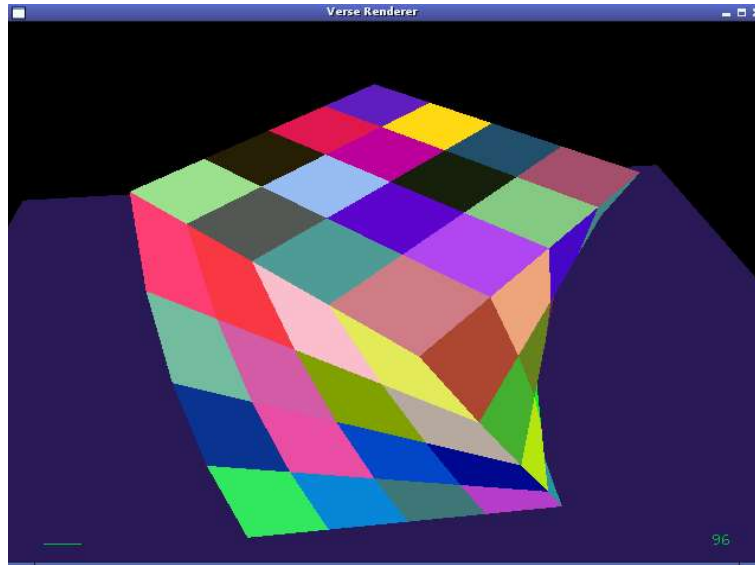


Illustration 1 Warped Cube Screen shot

This is a significant result, since it displays a working “construction history” pipeline, something the core Vere technology does not have. Still, the resulting data is successfully distributed using Verse, and made accessible to other clients that need to know nothing about how it was created.

The Purple API

This section talks about the Purple API, which is how plug-ins communicate with the engine in order to get work done. The API is in C, and defines four major categories of functions:

- Initialization; functions used from within, and only from within, a plug-in's `init()` function.
- Input-reading; functions used to read out the values (if any) on a plug-in's inputs.
- Node manipulation; functions used to create, destroy, and modify Verse node data. This is the largest category by a large margin.
- Output-writing; functions used to write values to a plug-in's output channels.

The plug-in programmer calls functions from these categories in order to achieve the desired results. The next sections describe each of the categories a bit further. Examples are given in the form of C code, to maximize clarity. The reader does not require a large degree of proficiency with C, it should be possible to glean a general sense of what is going on by just looking at the names of the variables and functions used.

Init-Functions

The functions in this category are fairly few (five) and are only used once in the running life time of a plug-in. They are very straight-forward, and typically only used inside the `init()` function of a plug-in. This function is the only externally visible function, and is called by Purple after the plug-in library has been loaded. Its task is to describe the code contained in the library, so that Purple can operate on it.

The smallest possible plug-in initialization goes like this⁵:

```
void init(void) {
    p_init_create("foo");
    p_init_compute(compute);
}
```

This tells Purple that the library contains one plug-in, called “foo”, and that the function to call to make that plug-in compute its value is called `compute()`. This of course assumes that such a function is also defined somewhere in the same library.

To specify that a plug-in has inputs (the above has none), another function call is needed in the initialization phase: Purple's `p_init_input()` function adds an input to a plug-in. Here's how using it looks during initialization:

```
void init(void) {
    p_init_create("bar");
    p_init_input(0, P_VALUE_BOOLEAN, "fun", P_INPUT_DONE);
    p_init_compute(compute);
}
```

The above adds an input named “fun”, which has boolean type (meaning it's either on or off). Further detail can be specified about a plug-in input by adding more code before the `P_INPUT_DONE` end marker.

There is also a call to add various meta-information to the plug-in's description. This information is published by Purple along with the more technical data, and can be used in interfaces to aid and inform users. Here's how adding author and usage texts looks like:

```
void init(void) {
    p_init_create("baz");
    p_init_compute(compute);
    p_init_meta("author", "Emil Brink");
    p_init_meta("desc/purpose", "Demo meta information");
}
```

The first parameter to `p_init_meta()` is called the “category”, and is a free-form text string that is used to group and identify the texts. It is not interpreted by the Purple engine, the pair `category=text` is simply published unmodified. It is assumed that there will be some “soft” standardization/common practice around this once Purple gains usage.

⁵ Plug-in code examples in this document are intended to communicate the “flavour” of the APIs, not to be a detailed programming reference. The code will not be described in full detail.

The fifth initialization call in Purple is used to define persistent state held by the Purple engine. It is not often used, and might be removed to simplify interfaces, so it has been omitted from this overview.

Input- Functions

There is a simple set of about 15 input-functions in the Purple API, that are used from within a plug-ins compute- function to retrieve current input values.

The reason for the seemingly large number of functions to deal with such a conceptually simple operation is C. As in, C does not support polymorphism, so a separate function is needed for each type of input value. While this is slightly troubling, it also makes the use of the functions very straight-forward and helps code readability a lot. Here's how retrieving a boolean input value looks, assuming the plug-in's compute- function is called `compute()` as in the examples above:

```
PComputeStatus compute(PPInput *input,
                      PPOutput output,
                      void *state) {
    boolean x;
    x = p_input_boolean(input[0]);
    /* More code follows here... */
}
```

Ignore the complicated-looking function signature for the moment, and concentrate on the single line of actual code inside the function. All the input-reading functions are named `p_input_TYPE()`, with `TYPE` chosen from the set of 15 supported types. A complete listing of these types is outside the scope of this document.

Node- Functions

The largest part of the Purple API by far is the set of functions that deals with node operations. This is simply because there are relatively many different types of nodes in the Verse data model, and Purple needs to support them all. The level of the support varies, but tries to be as complete as possible.

It would be very tiring to enumerate the full set of node functions in this document; there are around 120 in all at the time of writing. In general, the functions follow the Verse data model and network protocol fairly closely, but there are differences.

The most major difference is that since Purple buffers and mirrors all data in the background, plug-in programmers do not need to bother with network delays or round trips. To get at a geometry node vertex, a client programmer using the Verse API directly needs to:

1. Find the node ID of the interesting node
2. Subscribe to it, to learn ID of interesting layer
3. Once the layer is described, subscribe to it
4. Wait for the vertex position to be set

All of these operations have a network round trip in between, and are spread out in different places in the code due to the Verse API's reliance on callbacks to pass data to the application. In contrast, here is how to get the position of a vertex when writing a Purple plug-in:

1. Look up the node (by name, or from an input)
2. Look up the layer
3. Read out the position

These operations are immediate look-ups against Purple's local mirror of the Verse database, there are no round trips and they can be done in sequence in the same function. No callbacks need to be registered, in fact Purple does not use callbacks except for the main `compute()` entry point.

To be concrete, here is how the above would look in code, using the relevant parts of the Purple API:

```
PComputeStatus compute(PPInput *input,
                      PPOutput output,
                      void *state) {
    PINode *node;
    PNGLayer *layer;
    real64 x, y, z;

    node = p_input_node(input[0]);
    layer = p_node_g_layer_find(node, "vertex");
}
```

```

    p_node_g_vertex_get_xyz(layer, 0, &x, &y, &z);
    /* Do whatever with vertex position in (x,y,z). */
}

```

This snippet assumes the node arrives on the first input, and uses the relevant input call to get a pointer to the node. It then uses two of the geometry node calls (as indicated by the `_g_` part of the function names) to look up the layer, and then to get the vertex' position.

All Purple node functions check for invalid node pointers, so there is little need to test each return value in the plug-in.

Output- Functions

The output part of the API contains functionality for passing data on to the next plug-in in a graph, by sending it to the plug-in's output. As described above, every plug-in has a single output, capable of transferring a multitude of values.

The way to think of the output channels is like this: an output can hold a set of node data of any size, and one value for each of the 14 types of primitive values. So, it is possible to output a string and a boolean (primitive types), but not two different strings at the same time. It is also possible output a string and a boolean, four geometry nodes and a text node, or any other combination and amount of nodes.

For primitive types, there is no requirement that the values represent the same actual value, they are completely independent. So, if a plug-in is known to output an integer and a real number, and the integer is 42, one cannot assume that the real number is 42.0. The value is totally up to the programmer, Purple does not do any consistency checking or enforcing on them.

The reason for this freedom is that it is useful to be able to emit several different kinds of values, as the result of a single operation. It could be more flexible, without the limitation on just one value per primitive type, but that would introduce further complexity in identifying which value is which and so on. The current architecture is a compromise between expressive power and ease of implementation and use.

As an example of how the parallel values can be used, consider a plug-in that measures the dimensions of an input geometry node, and emits its volume. Such a plug-in could use a real number for the volume measurement, while emitting e.g. the number of inspected vertices as an integer. This allows the plug-in to solve two different but related problems at the same time.

To illustrate simple use of the output functions, here is part of a plug-in that adds to numbers together (as real values) and outputs their sum:

```

PComputeStatus compute(PPInput *input,
                       PPOutput output,
                       void *state) {
    real64 x = p_input_real64(input[0]),
           y = p_input_real64(input[1]);

    p_output_real64(output, x + y);
    return P_COMPUTE_DONE;
}

```

The above example should be fairly self-explanatory. It just reads out two reals from the first two inputs, and outputs their sum. The final `return`-statement tells Purple that the plug-in is finished.

There are several output-functions for primitive types, again because of the lack of polymorphism in C. There are also several for nodes, that are named slightly differently: output calls are used to create new node data as well as to just pass existing nodes along to the output.

The following snippet illustrates how to use the Purple output API to create a new object node, taking its name as an input string:

```

PComputeStatus compute(PPInput *input,
                       PPOutput output,
                       void *state) {
    const char *name;
    PNode *node;

    name = p_input_string(input[0]);
    node = p_output_node_create(output, V_NT_OBJECT, 0);
    p_node_set_name(node, name);
    return P_COMPUTE_DONE;
}

```

Again, this should be fairly self-explanatory, with one exception: the final argument to `p_output_node_create()`, the literal number zero. This number is an artifact of Purple's output caching system. It is called the “label”, and is imply an integer that uniquely identifies the node among the ones created by this plug-in. Labels start at zero and count upward with no holes or repeats, and are used internally in Purple to look up the created node in a cache, for quicker re-use between invocations.

This labeling system is not optimal, and might be replaced by some better scheme as time goes on and Purple gains more usage. For the moment, it is fairly minimal and works.

Purple Internals

Now that we've covered the main parts of Purple, it's time too look a little at how it is constructed internally. Representing the architecture graphically is difficult, but the figure below gives a rough outline of the relevant modules.

Abstract Data Types

Purple is written in C, which as a rather minimalist language does not have a standard set of abstract data types like lists, trees, and so on. While there are several such libraries available from various places, one goal when implementing Purple was to minimize the number of external dependencies. Therefore, Purple contains code for the necessary things, and does not rely on an external library. There is a fair chunk of such basic “infrastructure” code, around 30% of the total code size⁶.

There is little growth in the infrastructure code, and since most of it is fairly simple conceptually and lots of references exist, its size does not represent a proportional investment of development time.

In many cases, it is believed (although strict measurements have not been made) that time can be saved by implementing a custom data type rather than locating, integrating, and learning an existing implementation.

Node Database

As has probably been made obvious in the preceding sections, Purple contains a full mirror of the Verse data base. This means that it subscribes to everything on the host it is running against, and stores all the data. It then stays subscribed, updating the mirror as changes occur.

This means that a substantial amount—again, around 30%—of the code is devoted to this task. The Purple node database is not a good match for the existing Enough storage library, although using it was certainly considered. The main problem is that Enough does not expose the internals of its nodes, which is something Purple needs to have access to for synchronization purposes.

Purple implements the full Verse data model, with the only exception currently being audio streams. Since streams are very latency-sensitive, they have been omitted from the Purple system. It's possible that stream support will be added further on if it's deemed interesting. Audio storage (in audio buffers) is fully supported, so Purple does not ignore the audio node completely, just the streaming part of it.

Node Synchronization

The task of taking two Verse nodes and comparing them in order to make one equal to the other is common in Purple, and is called node synchronization. This is done whenever a node reaches an output plug-in, and needs to be sent to the Verse host. The output plug-in passes the nodes to the synchronizer, which adds them to its “work in progress”-list.

On each iteration of Purple's main loop, the synchronizer is given some CPU time to use. It will pick off a node from its work queue, look up which remote node represents that data, and start comparing the two. Comparisons are hard-coded and written specifically for each node type, as it's not possible to just compare the bits directly. As differences are found, the synchronizer generates and emits a Verse command to remove the difference. These commands always strive to make the remote (externally visible) node equal to the local one.

If the synchronizer's comparison fails to find any differences, Purple concludes that the node is in fact synchronized, and removes it from the work queue. If changes don't keep occurring (due to graph action), this will quickly end up with an empty queue, meaning Purple is again idle.

The synchronization algorithm, as it is, can be summarized like this:

⁶ In April 2005, this amounts to roughly 6,000 lines of code.

```

synchronize(nodelocal)
    if nodelocal has remote version noderemote
        while d is difference from nodelocal to noderemote
            send d to server
    else
        send node_create to server
        synchronize(nodelocal)

```

The techniques involved in node comparison are not very interesting, it's typically just iterating over a node's contents, looking it up in the target, and then comparing the "value" part of the data. Since Verse data is very finely addressable, it's often a 1:1 match between items that are compared and the resulting change-command that gets sent. For instance, if two vertices in a XYZ-type layer are differing, the update is simply a `g_vertex_set_xyz` command that overwrites the mismatched vertex with the source data. There is one exception to this kind of addressing granularity, though: the text node's buffers. Such buffers contain free-form text, with no other addressing than character-count index and length values. To compare these, Purple incorporates a diff-algorithm, which finds a minimal set of changes, which are then sent as `t_text_set` commands to insert or delete text as necessary.

API Implementation

The purpose of the API part of Purple's code is to implement all the calls defined in the public Purple API, and act as a glue between the "plug-in view" of the world and the Purple engine's view. This involves converting abstract node and layer references into actual pointers to internal node database data, and vice versa.

A large part of the API implementation is mere glue between the Purple API and the types used in the node databasing modules. This keeps the total for the actual API implementation at around 10% of the total Purple code size.

Graph Management

Around 10% of the code is devoted to graph management, and implements the in-memory representation of graphs, supports operations on graphs, and also is responsible for keeping the external representation updated by sending new XML fragments to the host as needed.

The graph module also creates the methods used to control Purple, and directly responds to any calls to them. Since this is the main way to control Purple, the graph module is not directly called by other parts of the Purple code very much.

Application Core

The remaining chunk of code all falls into a generic "application core" category, and deals with things that are required to make Purple into a coherent whole. Here we have the definition of the `main()` function that is the starting point for all C programs, containing various initializations. Also the program's main loop resides in this part of the code, as does the code needed to make the program into a Verse client.

Conclusions

The Purple project as a whole is hard to judge as a success or failure at present. Technically, there exists a code base that solves the problems we set out to solve. It is not easy to say if it solves them well enough, far less "correctly".

We are hoping to be able to build some kind of community interest in Purple, and to learn more about the design's good and bad points through more use of the code.

Purple is licensed under the GNU GPL, which makes it Open Source (and Free Software). The code is available from CVS⁷ for anyone to check out and/or browse on line.

During the upcoming couple of months, the Uni-Verse project is about to do its first organized "release" of software, including Purple, and we are hoping we will get more feedback from users as a result.

⁷ See <http://projects.blender.org/cvsx/?group_id=35>, module is "purple".

Appendix: Terminology

This section lists some terms that appear often when talking about Purple, and give definitions in that context. Terms are in alphabetical order.

engine

The core executable in the Purple system. The engine is the program that loads the plug-ins, publishes information about available plug-ins and graphs, and makes things happen.

graph

A set of modules. Strictly speaking, a Purple graph need not be connected at all, i.e. all the modules can be independent, or there can be multiple “sub-graphs” that do not share information among each other. Typically, a single graph should be used for a single logical task, and be fully connected.

library

A binary file containing computer code implementing one or more plug-ins. The distinction between a library and a plug-in is often omitted in this document, for brevity and simplicity. It exists mainly as a convenience for plug-in developers; sharing code between a set of related plug-ins is made easier by the possibility of writing several distinct plug-ins in a single C file, and compiling it into one library.

module

An instantiated plug-in. It is often necessary to distinguish between a plug-in in general, and a specific instance of that plug-in. This term provides that distinction. The word pair plug-in/module has much in common with the pair class/object as used in object-oriented programming.

plug-in

The main building block in Purple. A plug-in is a “black box” containing some kind of functionality. This functionality is typically controlled by a number of inputs that accept values, and results in (other) values appearing on the plug-in's output. An input can be connected to another plug-in's output, which is how graphs of complex functionality are built.

synchronization

The process of taking a locally created node, the result of a graph computation, and comparing it to a remote node that is that node's external representation. Any differences found are removed by making the remote node into a copy of the local one.