Purple

A Data-Flow-triggered, Plug-In-based Computational Engine using Verse

Emil Brink <emil@obsession.se>
Computational What?

• Interesting way to implement internals of traditional 3D creativity packages
  – Tools/Modifiers
  – Constraint Evaluation/Enforcement
• Very general, above is example, not full goal
• Focus on data processing, not I/O or storage
• Internals really separated from user interface
• Encourage distributed development
The Point?

• There are several, somewhat overlapping
• Implement the back-end of a 3D package, using Verse for the front-end
  − Reap Verse's benefits (networked, multi-user, ...)
• Allow building modeling application
  − Of interest as next-generation Blender?
• Improve Verse experience
• Provide a nice playground for programmers
  − Plug-ins easy to develop
  − Hoping for community
History

• Original concept from Eskil Steenberg, Verse's inventor
• Discussed with and judged as being of interest to Blender by Ton Roosendaal
  – Ton and Eskil met at SIGGRAPH 2002
• Implementation started Summer 2004, by me
  – As part of the Uni-Verse project, BF is partner
  – All code in CVS at blender.org
Plug-Ins

- Computation expressed by running plug-ins
- Aggregated by connecting plug-ins into *graphs*
- Set of inputs
  - Loosely typed
- Single logical output
  - Multiple independent values
- Reasonably easy access; better focus
- Heuristics allow this-to-that connections as often as possible
  - Avoid errors, let *something* happen
Examples

- **Modeling Primitives**
  - Easy to do parametric primitives
  - Purple does I/O and storage
  - One-parameter cube in ~60 lines of C

- **Bitmap Filters**
  - Reasonably direct access
  - Writing filters can be rewarding

- **Constraints, Combinatorial Logic, Tests**
  ("Intelligence")

- **Whatever else you feel like...**
Data Flows

• Plug-in runs when input(s) change
• Input can come from user, e.g. a constant
  – Create a red sphere...
  – ...with radius 4.711 units
• Input can come from other plug-in's output
  – Causes re-computation to flow through graph
  – Models construction history
Purple Engine

• Plug-ins run by Purple engine program
• Implemented as a Verse client!
  – Operates on shared data
  – Client/server distribution just happens
  – Opens up major sharing possibilities
• Mirrors all data
  – Hides network round trips when possible
  – Local data makes things easier
• Loads plug-ins, manages execution
No UI

• Purple Engine is pure computational code, no user interface
• Intended to be remotely controlled by dedicated UI clients
• Allows computation/tools to be developed separately from UI
• No such client exists today :/
  – Idea: incorporate Purple control into Connector
Controlling Purple

• Separating out the UI places requirements on Purple:
  – Publish a list of available plug-ins
  – Publish a list of existing graphs
  – Graph descriptions
  – Support graph editing

• Information published using Verse standard text node, holding XML data
  – Very detailed, allows UI to do clever things

• Graph descriptions are also XML

• Graph editing through Verse avatar methods
System View

- Plug-ins run in Purple process
- Links use the Verse protocol to communicate
- Purple engine runs close to host for network performance
- Clients can run wherever
Plug-In Programming

• A C API, defined in single header purple.h
  – Verse core currently required (verse.h)
• Four distinct classes of calls:
  – Init – describe plug-in to the engine core
  – Input – read out input values
  – Node – work with various aspects of Verse nodes
  – Output – cause data to appear at output
• Initialization done once; sets compute callback
  – More (meta, state, N plug-ins per library, ...)
• Engine calls compute() when inputs change
More on Computation

- Plug-in code is run in a cooperative multitasking fashion; never interrupted by Purple
  - Forces plug-in programmer to think a little
  - This might change in future versions :)
  - Parallel execution of plug-ins is desired, too
  - Plugin-returns “done” or “call again soon”
- The Purple engine knows what is connected to each input, and can run `compute()` when something changes
  - Coarse filtering, but hopefully good enough
Plug-In Code Example

```c
#include "purple.h"

static PComputeStatus compute(PPInput input[], PPOutput output, void *state)
{
    real32 x = p_input_real32(input[0]), /* Read out inputs. Heuristics. */
               y = p_input_real32(input[1]);

    p_output_real32(output, x + y); /* Emit sum as real32 result. */
    return P_COMPUTE_DONE; /* We're done for this time. */
}

void init(void) /* Purple runs this when plug-in loads. */
{
    p_init_create("add-as-real32"); /* User-visible plug-in name. */
    p_init_input(0, P_VALUE_REAL32, "x", P_INPUT_DONE); /* Two inputs. */
    p_init_input(1, P_VALUE_REAL32, "y", P_INPUT_DONE);
    p_init_compute(compute);
}

• API tries hard to be clean
• P_ prefix used for Purple constants, types, and functions
• Actual compute() function is basically a single statement, x+y
• Inputs are referenced by index, init-name is for UI uses
```
About `compute()`

- The `compute()` callback:
  ```c
  PComputeStatus compute(PPInput input[],
                         PPOutput output,
                         void *state);
  ```

- Gets array of inputs
  - As many as created w/ `p_init_input()`
  - The type is “Purple Port, Input”

- Gets single output port

- Gets state pointer (managed by Purple)

- Returns done/not done status
Inputting Nodes

- Plug-in might require a node as its input
- Kind of problematic, design-wise
  - What to connect to input in UI?
  - Need explicitness for dependency tracking
- Solved using built-in plug-in, “plugin-input”
  - Has string input, outputs node of that name
  - Magically runs `compute()` not only when string changes, but when named `node` changes, too!
  - Gives UI something known to manage
#include “purple.h”

static PComputeStatus compute(PPInput input[], PPOutput output, void *state)
{
    PINode *node = p_input_node(input[0]);  /* Get node, if available. */

    if(node != NULL)
        p_output_uint32(output, strlen(p_node_name_get(node)));
    return P_COMPUTE_DONE;
}

void init(void)
{
    p_init_create(“name-length”);  /* User-visible plug-in name. */
    p_init_input(0, P_VALUE_MODULE, “node”, P_INPUT_DONE);  /* Must be “internal”. */
    p_init_compute(compute);
}

• P_VALUE_MODULE signals to UI that the plug-in needs to be connected to the output of another plug-in, such as plugin-input
• The test in compute() protects libc’s strlen() from NULL
• It is fine to skip output if conditions aren’t right
#include “purple.h”

static PComputeStatus compute(PPInput input[], PPOutput output, void *state)
{
    const real32 *v1 = p_input_real32_vec3(input[0]),
        *v2 = p_input_real32_vec3(input[1]);

    p_output_real32(output, sqrtf(powf(v1[0] – v2[0], 2.0f) +
                                powf(v1[1] – v2[1], 2.0f) +
                                powf(v1[2] – v2[2], 2.0f));

    return P_COMPUTE_DONE;
}

void init(void)
{
    p_init_create(“vec3-dist”);
    p_init_input(0, P_VALUE_REAL32_VEC3, “v1”, P_INPUT_DONE);
    p_init_input(1, P_VALUE_REAL32_VEC3, “v2”, P_INPUT_DONE);
    p_init_compute(compute);
}

• Above can be used to measure distance between object nodes
• No need to explicitly input nodes; heuristics help here
Using vec3-dist

- The plugin-input instances output nodes
- `vec3-dist` reads `real32_vec3's`
- Heuristics convert nodes into `vec3`
  - For object nodes, position is returned
  - Also possible to be more explicit
Explicit Node Access

```c
#include "purple.h"

static PComputeStatus compute(PPInput input[], PPOutput output, void *state) {
    PINode *n1 = p_input_node(input[0]), *n2 = p_input_node(input[1]);

    if(p_node_type_get(n1) == V_NT_OBJECT && p_node_type_get(n2) == V_NT_OBJECT) {
        const real32 *v1 = p_node_o_pos_get(n1), v2 = p_node_o_pos_get(n2);
        p_output_real32(output, sqrtf(powf(v1[0] - v2[0], 2.0f) +
                                 powf(v1[1] - v2[1], 2.0f) +
                                 powf(v1[2] - v2[2], 2.0f));
    }
    return P_COMPUTE_DONE;
}

void init(void) {
    p_init_create("obj-dist");
    p_init_input(0, P_VALUE_MODULE, "n1", P_INPUT_DONE);
    p_init_input(1, P_VALUE_MODULE, "n2", P_INPUT_DONE);
    p_init_compute(compute);
}
```

- Explicitly access object nodes, testing to make sure
- Could also just output the position, and “graph it”
- Purple API uses Verse constants when convenient
More on Outputs

- Each plug-in has a single output port, passed to `compute()`
- Use `p_output_XXX()` to pass along values
- You can pass distinct values for the 9 simple types (e.g. integer = -12, uint32 = 56, ...)
  - Useful for heuristics and additional data
- You can also pass a (set of) full Verse nodes
```c
#include "purple.h"

static PComputeStatus compute(PPInput input[], PPOutput output, void *state) {
    real32 x = p_input_real32(input[0]), y = p_input_real32(input[1]);

    if(x > y) {
        p_output_boolean(output, TRUE);
        p_output_string(output, "yes");
    }
    else {
        p_output_boolean(output, FALSE);
        p_output_string(output, "no");
    }
    return P_COMPUTE_DONE;
}

void init(void) {
    p_init_create("greater-than");
    p_init_input(0, P_VALUE_REAL32, "x", P_INPUT_DONE);
    p_init_input(1, P_VALUE_REAL32, "y", P_INPUT_DONE);
    p_init_compute(compute);
}

• Emit both boolean and textual results
```
Accessing Nodes

- There are many calls for working with nodes
  - Follows Verse, `p_node_g_vertex_set_xyz()`
  - Will have more luxurious calls later on
    - Take advantage of locality of data set
    - Need more experience/pressure/time/ideas

- Nodes represented as opaque pointers
  - `PINode*` (input, read-only)
  - `PONode*` (output, read/write)

- Magic happens behind the scenes
  - Output synchronizer/command generator
Node Changes

- Changing node state adds to sync. queue
- Compared to input version, if available
  - Comparison generates commands (patches)
  - Utilizes XML version of the protocol spec, xslt
  - Commands sent will be returned by Verse host, node updated → fewer differences → done
  - Very expensive for structural changes... 😞
- Creating new node adds it to special queue
  - Node is created, then synced as usual
Purple Status

- Been under way since May
- Single developer (me)
- No hard specification, evolves
- Code base is roughly 10,500 lines of C
  - Includes utility code (data structures)
  - Full rebuild in ~10 seconds on my puny laptop
- Lots of things still to do
  - Flesh out node API
  - Write synchronizer
  - ...
Questions?

Let's hear them!

mailto:emil@obsession.se
http://www.blender.org/modules/verse/