PhaseSpace Impulse: Software API

Document Purpose

This document describes the API (Application Programming Interface) for the PhaseSpace Impulse system. This document, along with the PhaseSpace software development kit (SDK), will be useful for users who wish to develop their own custom client applications. It is through such custom clients, using the PhaseSpace API, that the customer can command and receive data from the system.

PhaseSpace client software is described in the following document:

- ***PhaseSpace Impulse: Software***

Additional information about the PhaseSpace system can be found in the following documents:

- ***PhaseSpace Impulse: Overview***
- ***PhaseSpace Impulse: Camera System***
- ***PhaseSpace Impulse: LED System***
- ***PhaseSpace Impulse: Configuration Manager***

Table of Contents

1: Overview

The PhaseSpace Impulse system captures complex motion in real time using advanced hardware and software technology. Motion capture is accomplished by placing the PhaseSpace cameras around a capture volume, and moving objects with LEDs attached to them.

The cameras detect the positions of the LEDs and transmit this information to a central computer that processes the data and calculates actual positions. These positions are then available for further processing by client systems in a client server environment.

The PhaseSpace system consists of:

- Cameras
- LED Base Station
- LED Driver Unit(s)
- LEDs
- A HUB into which the cameras and the LED Base Station connect
- A server computer which runs Linux and communicates with the HUB
- Calibration object(s)
- Server and client software
- Dynamic link libraries that enable a user to construct custom client programs.

2: Hardware and System Software

The PhaseSpace server software runs on the PhaseSpace server machine. Client software (PhaseSpace clients or custom user clients) can run on Linux or Windows machines.

PhaseSpace delivers the server machine fully configured with PhaseSpace server and client software preinstalled. Separate Linux and Windows client software packages are also included with the PhaseSpace Impulse system. Separate client machines are optional. The minimum requirements of the hardware and system software configurations are detailed below.

2.1 PhaseSpace Server

Server machines are only provided by PhaseSpace (customers cannot purchase components for creating their own servers). The server configuration is listed below for reference.

- Intel Pentium IV 3 GHz or AMD Athlon XP 4000+
- 512 MB Memory
- CDROM
- Keyboard
- Mouse
- Monitor
- USB Port
- Graphics Accelerator Card
- Gigabit network
- Slackware 10.1 (Linux kernel v.2.6.15)
- PhaseSpace HUB
- PhaseSpace Server Software.

2.2 Client Linux

Linux client machines may be obtained directly from PhaseSpace, but users normally purchase client machines from outside sources. Client Linux boxes should meet the following minimum specifications.

- Intel Pentium IV 2 GHz or AMD Athlon XP 2000+
- 512 MB Memory
- CDROM
- Keyboard
- Mouse
- Monitor
- Graphics Accelerator Card
- Gigabit network
- Linux (kernel v.2.6.15)
- PhaseSpace Linux client software

2.3 Client Windows

Windows client machines may be obtained directly from PhaseSpace, but users normally purchase client machines from outside sources. Client Windows boxes should meet the following minimum specifications.

- Intel Pentium IV 2 GHz or AMD Athlon XP 2000+
- 512 MB Memory
- CDROM
- Keyboard
- Mouse
- Monitor
- Graphics Accelerator Card
- Gigabit network
- Windows XP
- PhaseSpace Windows client software

3: Software API

The server software consists of two components:

- The core software that handles communication over the USB port between the computer and the PhaseSpace Impulse System.
- The socket library that handles communication between client machines and the server.

The client software must link to the PhaseSpace library that exposes the API.

3.1 Initialization

3.1.1 Server Connection

This following is the main system initialization routine. It connects the client to the core routines of the server. It opens a socket and configures the communication channels to pass data for the PhaseSpace Impulse System. This function will block until there is a connection or an error.

```
int owlInit(const char *server, int flags);
```

Connects to a server computer specified by *server*, default is “localhost”. The *flags* argument is a bit wise logical OR combination of the following values:

OWL_SLAVE	Connects to *server* in slave mode. Additional clients after the primary client should use this flag.
OWL_POSTPROCESS	Turns on extra filtering and recovery, but adds latency and CPU usage
OWL_MODE1	Set system to Mode 1
OWL_MODE2	Set system to Mode 2

OWL_MODE3	Set system to Mode 3
OWL_MODE4	Set system to Mode 4
OWL_CALIB	Set system to calibration mode.
OWL_CALIBFINE	Set system to fine calibration mode.

Returns the passed *flags* if OK, less than 0 on error.

Only one primary client is allowed to connect to the server, while multiple slave clients are permissible. Only the primary client is allowed to specify initialization flags. All connections to the server are closed once the primary client is terminated. On the other hand, a slave client is only able to close its own connection.

3.1.2 System Cleanup

The following performs the system cleanup necessary before client termination.

```
void owlDone(void);
```

3.2 Client Side Setup

Transformations of standard data to user specific data are done by the client. The transformation parameters must be set before any operations (other than initialization) can be performed. The following transformations are available.

3.2.1 Scale Factor

The scale factor is set with the following:

```
void owlScale(float scale);
```

This sets the scale factor for the user space. Data from the Server is multiplied by the scale factor, before being returned to the user. Scale is also applied during rigid body creation. owlScale() must be called **after** constructing a rigid body.

3.2.2 Pose

The Pose the user wants is set with the following:

```
void owlLoadPose(const float *pose);
```

This transforms the default data from being Camera 0 oriented, to being oriented according to the way the user wishes to view the data. For instance, one may want to view the data from a position directly in front of and above Camera 0 looking down. By using owlLoadPose, the user is able to construct a pose that would do this transformation.

A pose consists of seven float values: vector (x, y, z) and a quaternion (s, x, y, z). This is basically the offset position from camera 0 for a pseudo camera and the rotation of that pseudo camera.

The unity pose (just use camera 0) is (0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0).

3.3 Server Side Setup

The client can configure certain parameters on the server at run time; the following routines are used to send data to the server in order to perform this configuration.

3.3.1 Integers

Integers are passed to the server with two routines. The difference between the routines is whether the integer data is direct or via a pointer.

```
void owlSetInteger(OWLenum pname, int param);
void owlSetIntegerv(OWLenum pname, const int *param);
```

Acceptable *pname* arguments:

OWL_STREAMING	The associated *param* values OWL_ENABLE or OWL_DISABLE indicate whether to enable or disable streaming of data from the server. This setting affects only the current client.
OWL_BUTTONS	The associated *param* values OWL_ENABLE or OWL_DISABLE indicate whether to enable or disable streaming of buttons. This setting affects only the current client.

OWL_MARKERS The associated *param* values OWL_ENABLE or OWL_DISABLE indicate whether to enable or disable streaming of markers. This setting affects only the current client.

OWL_INTERPOLATION Enables interpolation and sets maximum interval to be interpolated (*param* must be a positive value). The default value is 0, which indicates that interpolation is disabled.

3.3.2 Floats

Floating-point values are passed to the server with two routines. The difference between the routines is whether the floating-point data is direct or via a pointer. The two calls are:

```
void owlSetFloat(OWLenum pname, float param);
void owlSetFloatv(OWLenum pname, const float *param);
```

Acceptable *pname* argument:

OWL_FREQUENCY This specifies the rate at which the server streams data (affects all clients). The associated *param* must be between 0.0 and OWL_MAX_FREQUENCY. Default value is 0.0, which effectively disables streaming. Setting frequency also enables streaming. If unsure of value to use for *param,* use OWL_MAX_FREQUENCY.

3.4 Trackers

Trackers are groups of related markers. There are two types of trackers, point trackers and rigid body trackers. Trackers can be created, destroyed, enabled, and disabled.

3.4.1 Create a Tracker

Trackers of either type (point or rigid body) are created with the following routines:

```
void owlTrackeri(int tracker, OWLenum pname, int param);
void owlTrackeriv(int tracker, OWLenum pname, int *param);
```

The parameter *pname* with the value OWL_CREATE creates a specified *tracker*. The argument *param* indicates the type of tracker to create: OWL_POINT_TRACKER or OWL_RIGID_TRACKER.

3.4.2 Enable, Disable, or Destroy a Tracker

Trackers are enabled, disabled, or destroyed with the following routine:

```
void owlTracker(int tracker, OWLenum pname);
```

The *pname* operation on the *tracker* can be OWL_ENABLE, OWL_DISABLE, or OWL_DESTROY.

3.4.3 Unused Tracker Routines

The following calls are available to pass values to trackers.

```
void owlTrackerf(int tracker, OWLenum pname, float param);
void owlTrackerfv(int tracker, OWLenum pname, const float *value);
```

The above routines pass either a single floating-point value or a set of floating point values to the specified tracker. Single values may be passed directly, while multiple values my only be passed through the pointer. When using owlTrackerfv, the pname/operation defines the number of values in the set, but since no operations are currently defined, for either of these routines they are unused.

Acceptable *pname* argument:

OWL_SET_FILTER Sets the Kalman filter parameters. Currently only supported for the rigid body tracker (4 values). Advanced knowledge of the Kalman filter is necessary in order to specify the correct parameters.

3.5 Markers

Once the Tracker has been defined, the LEDs or **Markers** that the tracker/system will be following need to be defined. Additionally, if the tracker is a rigid body tracker, the relative positions of the markers need to

be specified. Markers CANNOT be shared between active trackers. Finally the Marker can be cleared from the system once its use is over. The following routines are used for these purposes.

Since markers are related to trackers, it is necessary to use the **MARKER** macro in all of the following routines to create/obtain the *marker* ID.

3.5.1 MARKER Macro

The **MARKER** macro takes a *tracker* and an *index* and produces a *marker* ID. The index is generally a sequential number sequence starting at 0, with a new number for each marker defined. Remember that the **MARKER** macro merely produces a *marker* ID. The **MARKER** macro is invoked as follows:

```
MARKER(tracker, index)
```

3.5.2 Set a Marker

The following routines are used to set a marker:

```
void owlMarkeri(int marker, OWLenum pname, int param);
void owlMarkeriv(int marker, OWLenum pname, int *param);
```

The parameter *pname* with the value of OWL_SET_LED sets the specified *marker* to an LED ID specified by *param*. The *marker* ID is obtained by using the **MARKER** macro.

3.5.3 Define Relative Positions of Rigid Body Marker

A rigid body is defined a set of markers where the positions of the markers are fixed relative to each other. We use the following routine to set the positions of the markers for a rigid body.

```
void owlMarkerfv(int marker, OWLenum pname, const float *param);
```

The parameter *pname* with the value OWL_SET_POSITION sets the relative position of the specified *marker*. The *marker* ID is obtained using the **MARKER** macro. The parameter *param* specifies the coordinates (x, y, z) of the *marker*.

3.5.4 Clear a Marker

Markers are cleared with the following routine:

```
void owlMarker(int marker, OWLenum pname);
```

The parameter *pname* with the value OWL_CLEAR_MARKER clears the specified *marker*. The *marker* ID is obtained using the **MARKER** macro.

3.5.5 Unused Marker Routines

The following Marker routine is defined but currently unused.

```
void owlMarkerf(int marker, OWLenum pname, float param);
```

3.6 Status and Errors

In most cases routines do not block waiting for a return value, therefore it is the user's responsibility to periodically check the system status and errors.

3.6.1 Status

Checking system status will block until all commands are processed and any errors are sent to the client.

```
int owlGetStatus(void);
```

Returns 1 is status is good (no errors), 0 if errors are present.

3.6.2 Errors

Errors that occur during run time are placed in a FIFO. Since it is possible to get multiple errors generated at a time, the FIFO should be emptied by multiple queries before proceeding. The following routine gets a value from the FIFO.

```
int owlGetError(void);
```

Possible return values are:

OWL_NO_ERROR	No errors at this time
OWL_INVALID_VALUE	Value passed was invalid or out of range
OWL_INVALID_ENUM	Enumeration passed was invalid
OWL_INVALID_OPERATION	Operation requested was invald

3.7 Obtaining Data

Once the system has been initialized and the trackers and markers defined, position data may be obtained from the core.

3.7.1 Camera Data

The camera data consists of a camera id; the pose of the camera, relative to camera 0; and a conditional that represents the confidence factor. Camera data is invariant during a user's run and only changes from calibration to calibration. The routine used to get the camera data is:

```
int owlGetCameras(OWLCamera *cameras, size_t count);
```

This will attempt to fill *count* OWLCamera structures pointed to by *cameras*. The returned value is the actual number of OWLCamera structures filled (if greater than or equal to 0). If the return value is less than zero this represents an error.

3.7.2 Marker Data

Point tracker data is obtained with the following call:

```
int owlGetMarkers(OWLMarker *markers, size_t count);
```

Where *markers* is a pointer to an array of *count* OWLMarker structures.

The return value is an error if less than 0, or if greater than 0, the number of structures filled.

The OWLMarker structure contains an integer id number; a frame number, which is a monotonically increasing integer determined by the system; three floats representing the x, y, and z coordinates of the marker; and a float representing the validity of the data, with numbers less than zero

being bad and numbers greater than zero being the confidence of the correctness of the x, y, z position.

3.7.3 Rigid Body Data

Obtaining Rigid Body Data is similar to obtaining point tracker data. The main differences are that the data structure contains pose data instead of simple x, y, z coordinates and the call is different. The call is:

```
int owlGetRigids(OWLRigid *rigid, uint_t count);
```

Where *rigid* is a pointer to an array of *count* OWLRigid structures.

The return value is an error if less than 0, or if greater than 0, the number of structures filled.

The OWLRigid structure contains an integer id number; a frame number, which is a monotonically increasing integer determined by the system; a pose, which is seven floats where the first three floats represent the x, y, and z coordinates of the rigid body, then 4 floats representing a quaternion (s, x, y, z) of the rigid body; the final float in the structure represents the validity of the data, with numbers less than zero being bad and numbers greater than zero being the confidence of the correctness of the pose.

3.7.4 Integers

Integers and arrays of integers are obtained with the following call:

```
int owlGetIntegerv(OWLenum pname, int *param);
```

Where *param* is a pointer to an array of ints.

The return value is an error if less than 0, or if greater than 0, the number of integers filled.

Acceptable *pname* arguments:

OWL_STREAMING	Obtains the state of streaming. Values of the returned integer are either OWL_ENABLE or OWL_DISABLE.
OWL_BUTTONS	Obtains the state of button streaming. Values of the returned integer are either OWL_ENABLE or OWL_DISABLE.

OWL_MARKERS	Obtains the state of marker streaming. Values of the returned integer are either OWL_ENABLE or OWL_DISABLE.
OWL_INTERPOLATION	Obtains interpolation interval value.
OWL_FRAME_NUMBER	Obtains current frame number.

3.7.5 Floats

Floats and arrays of floats are obtained with the following call:

```
int owlGetFloatv(OWLenum pname, float *param);
```

Where *param* is a pointer to an array of floats.

The return value is an error if less than 0, or if greater than 0, the number of floats filled.

Acceptable *pname* argument:

OWL_FREQUENCY	Obtains the current frequency of streaming.

3.7.6 Strings

Strings are obtained with the following call:

```
int owlGetString(OWLenum pname, char *str);
```

Where *str* is a pointer to an array of chars.

The return value is an error if less than 0, or if greater than 0, the number of chars filled, including the terminating null character.

The *pname* argument can be **OWL_VERSION**, which indicates the version of the OWL library.

A: Sample Programs

Sample code is listed below that demonstrates the use of the PhaseSpace API for point tracking and rigid body tracking. The actual C++ files are included with the software distribution in the **phasespace/src/examples** directory (Linux server), and are also included in the Windows PhaseSpace SDK distribution.

Example 1: Sample code for point tracking

```
// example1.cc
// simple point tracking program

#include <stdio.h>

#include "owl.h"

// change these to match your configuration

#define MARKER_COUNT 4
#define SERVER_NAME "localhost"
#define INIT_FLAGS 0

void owl_print_error(const char *s, int n);

int main()
{
  OWLMarker markers[32];
  int tracker;

  if(owlInit(SERVER_NAME, INIT_FLAGS) < 0) return 0;

  // create tracker 0
  tracker = 0;
  owlTrackeri(tracker, OWL_CREATE, OWL_POINT_TRACKER);

  // set markers
  for(int i = 0; i < MARKER_COUNT; i++)
    owlMarkeri(MARKER(tracker, i), OWL_SET_LED, i);

  // activate tracker
  owlTracker(tracker, OWL_ENABLE);

  // flush requests and check for errors
  if(!owlGetStatus())
    {
```

```
      owl_print_error("error in point tracker setup", owlGetError());
      return 0;
    }

  // set default frequency
  owlSetFloat(OWL_FREQUENCY, OWL_MAX_FREQUENCY);

  // start streaming
  owlSetInteger(OWL_STREAMING, OWL_ENABLE);

  // main loop
  while(1)
    {
      int err;

      // get some markers
      int n = owlGetMarkers(markers, 32);

      // check for error
      if((err = owlGetError()) != OWL_NO_ERROR)
      {
        owl_print_error("error", err);
        break;
      }

      // no data yet
      if(n == 0)
      {
        continue;
      }

      if(n > 0)
      {
        printf("%d marker(s):\n", n);
        for(int i = 0; i < n; i++)
          if(markers[i].cond > 0)
            printf("%d) %f %f %f\n", i, markers[i].x, markers[i].y,
markers[i].z);
        printf("\n");
      }
    }

  // cleanup
  owlDone();
}

void owl_print_error(const char *s, int n)
{
  if(n > 0)
    switch(n) {
    case 0: printf("%s: No Error\n", s); break;
    case 1: printf("%s: Invalid Value\n", s); break;
    case 2: printf("%s: Invalid Enum\n", s); break;
    case 3: printf("%s: Invalid Operation\n", s); break;
    default: printf("%s: 0x%x\n", s, n); break;
```

```
    }
  else printf("%s: %d\n", s, n);
}
```

Example 2: Sample code for rigid body tracking

```
// example2.cc
// simple rigid body tracking program

#include <stdio.h>

#include "owl.h"

// change these to match your configuration

#define MARKER_COUNT 4
#define SERVER_NAME "localhost"
#define INIT_FLAGS 0

float RIGID_BODY[MARKER_COUNT][3] = {
  { 0, 300,  0},
  { 300, 0,  0},
  {-300, 0,  0},
  { 0,  76, -200}
};

void owl_print_error(const char *s, int n);

int main()
{
  OWLRigid rigid;
  OWLMarker markers[32];
  int tracker;

  if(owlInit(SERVER_NAME, INIT_FLAGS) < 0) return 0;

  // create rigid body tracker 0
  tracker = 0;
  owlTrackeri(tracker, OWL_CREATE, OWL_RIGID_TRACKER);

  // set markers
  for(int i = 0; i < MARKER_COUNT; i++)
    {
      // set markers
      owlMarkeri(MARKER(tracker, i), OWL_SET_LED, i);

      // set marker positions
      owlMarkerfv(MARKER(tracker, i), OWL_SET_POSITION, RIGID_BODY[i]);
    }

  // activate tracker
```

```
owlTracker(tracker, OWL_ENABLE);

// flush requests and check for errors
if(!owlGetStatus())
  {
    owl_print_error("error in point tracker setup", owlGetError());
    return 0;
  }

// set default frequency
owlSetFloat(OWL_FREQUENCY, OWL_MAX_FREQUENCY);

// start streaming
owlSetInteger(OWL_STREAMING, OWL_ENABLE);

// main loop
while(1)
  {
    int err;

    // get the rigid body
    int n = owlGetRigids(&rigid, 1);

    // get the rigid body markers
    //  note: markers have to be read,
    //  even if they are not used
    int m = owlGetMarkers(markers, 32);

    // check for error
    if((err = owlGetError()) != OWL_NO_ERROR)
    {
      owl_print_error("error", err);
      break;
    }

    // no data yet
    if(n == 0)
    {
      continue;
    }

    if(n > 0)
    {
      printf("%d rigid body (%d markers):\n", n, m);
      if(rigid.cond > 0)
        {
          for(int i = 0; i < 7; i++)
          printf("%f ", rigid.pose[i]);
          printf("\n");
        }
      printf("\n");
    }
  }

// cleanup
```

```
  owlDone();
}

void owl_print_error(const char *s, int n)
{
  if(n > 0)
    switch(n) {
    case 0: printf("%s: No Error\n", s); break;
    case 1: printf("%s: Invalid Value\n", s); break;
    case 2: printf("%s: Invalid Enum\n", s); break;
    case 3: printf("%s: Invalid Operation\n", s); break;
    default: printf("%s: 0x%x\n", s, n); break;
    }
  else printf("%s: %d\n", s, n);
}
```

B: Glossary

API

Application Program Interface. This is the standardized way of accessing the PhaseSpace Motion Digitizer System.

FIFO

Short for First In First Out: A mechanism for ordering data for a user who would want the data in the sequence that it was generated.

LED

Light Emitting Diode

LIFO

Short for Last In First Out: A mechanism for ordering data for a user who would want the data in the sequence from most recently to least recently generated. Also known as a stack.

MARKER

Where the system thinks that the LED is.

OS

Operating System.

OWL

Name originally given to the PhaseSpace Motion Digitizer System since the early cameras looked like an owl with two beady eyes.

QUATERNION

The method of specifying rotations and orientations of coordinate systems. At PhaseSpace, quaternions encode spatial rotations by four real numbers, analogous to the familiar Euler angles, such as yaw, pitch, and roll. Quaternions play an essential role in the representation of object rotations in computer graphics, primarily for animation and user interfaces. Quaternions occupy a smooth, seamless, isotropic space, which is a generalization of the surface of a sphere. Thus, there is no need for special care to avoid singularities, such as gimbal lock, where two rotation axes collapse into one and thus make the interpolation irreversible.

TRACKER

A group of (related) MARKERs.

Technical Support

For any questions regarding hardware, software, or documentation use the contact details given below to contact PhaseSpace.

1933 Davis Street, Suite 294,
San Leandro, CA USA 94577
(510) 638-5035
www.phasespace.com

Technical Support: support@phasespace.com