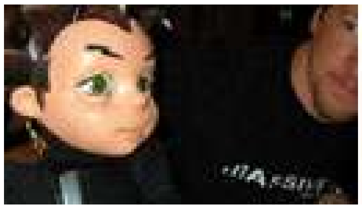
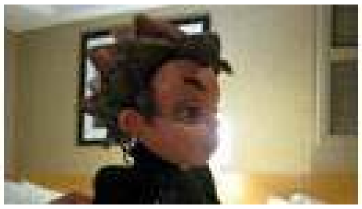
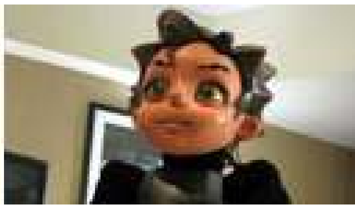
RECURSION SOFTWARE, Inc.

Real-time Robot Animation: Building the Skeleton of a Software Brain using Massive, Maya and a Real-Time C++ Framework

By Bob Hauser, Director of Software
Hanson Robotics



Abstract

Adjusting to continually changing requirements is a real problem developers face. This challenge can be overcome by careful planning, which includes finding the right tools that provide flexibility and extensibility. Read this case study to learn how Hanson Robotics delivers cutting edge robots by leveraging software tools to develop a real-time C++ robot control framework and successfully respond to changes along the way. Lessons learned include how to deal with changing requirements due to legacy code, integration issues, performance problems, and dependencies on 3rd party code.

Contents

Overview 2
The Shape of Things to Come 2
Background: Robotics 2
Finding Shoulders to Stand On 3
Software Selection Criteria 3
Growing Pains 3
Instability 3
Customizing Massive 4
Performance Issues 4
Porting to Windows 4
Initial Results 5
Crawl, Walk, Run 5
Listing 1 – Example wait for multiple objects 6
Listing 2a – A sample streaming class header 7
Listing 2b – A sample streaming class 7
Listing 3 – Implementation 1. Writes to the stream are to the underlying socket. Toggle the Nagle behavior to flush out a packet. 8
Listing 4 – Implementation 2. Using os_cache to buffer writes to the stream until the cache is flushed. 8
About the Author 8
About Recursion Software, Inc. 9

Overview

Writing software can be a joy for developers who undertake their craft to solve research problems and meet business requirements. Unfortunately, the original scope of the problem almost always changes as the development team is implementing the project. Choosing flexible tools at the outset allows the development team to successfully adapt to changes in requirements.

In this paper, we will look in some detail at an ongoing project to develop a robot control framework that supports state-of-the-art animation tools and provides a foundation for building a cognitive architecture with real-time processing needs. Along the way we will encounter continually changing requirements inherent to working with legacy code, integration issues, performance problems, and dependencies on 3rd party code that has its own independent release schedules.

The Shape of Things to Come

At Hanson Robotics, a leading robotics company, a project to develop the next generation robot control software platform was underway. The problem is broad in scope — build the software framework and components for real-time computer vision analysis, speech recognition and synthesis, cognitive-emotive decision making, and bring the best 3D animation tools into the robot authoring and robot runtime environments. For this article we restrict our focus to the framework development and integration with key components.

Background: Robotics

Lifelike robotics heads come with a diverse set of independent facial controls that allow for a full range of facial expressions. The fine-grained control creates a unique problem in robotics development — how to effectively blend facial expressions in real-time driven by various sources including authored content, internal cognitive states, synthesized speech, and external sensor events. Solving the real-time expression blending led to integrating leading 3D animation tools to facilitate the process.

As with most real world software development projects, this task brings additional complexities of needing to incrementally improve legacy code and deal with changing requirements along the way.

Finding Shoulders to Stand On

While certain low-level processing is done on board the robot, the majority of robot control software is run on a connected computer. One key early requirement was the stabilization and improvement of the existing implementation which consisted of multithreaded C++ and C# software. Threading problems and performance problems needed immediate attention. After evaluating the existing code it became clear we could benefit from a solid software framework. Due to our expected performance and integration needs we wanted to improve the C++ layer with a good cross-platform framework and migrate the C# components to Java.

Software Selection Criteria

For selecting C++ software we used the following criteria: readiness, low learning curve, stable, distributed, real-time, flexible, cost effective.
Readiness equates to how quickly the selected software can be deployed in a way that solves current problems. Whether commercial or open-source, a tool that requires a wholesale changeover impacting many functioning components at the same time is a common choice, but a poor one. We needed a tool that allowed us to immediately employ some benefits within the legacy software and eventually grow to use more. Having a low learning curve differs from readiness in the ease of acquiring new benefits. Important questions we asked included, is the software clearly documented? Does it have expected abstractions and patterns? Are there clear examples?

Stability criteria includes whether the product is proven and has had established real-world deployments. Good support for writing distributed applications was also necessary since some software components will be server-based.

The nature of robotics requires high performance from all software tools. Hard real-time processing is limited to on robot firmware but soft real time needs are pervasive. For life-size robot heads prompt face detection is needed and for biped robots gyro data becomes useless if the response latency grows too long.

Since we will see some of these requirements change over time, our tools should be flexible and cost-effective.

Growing Pains

After selecting software to accelerate developing our robot control framework, Recursion Software's C++ Toolkits™ in our case, we needed to improve performance around the legacy threading architecture. Fast firing timers were running in threads regardless of whether new work had arrived or not. Once we switched to event based processing the problem arose of how to cleanly shutdown threads. We created a separate event for thread termination and used a cross-platform wait for multiple objects abstraction (See Listing 1).

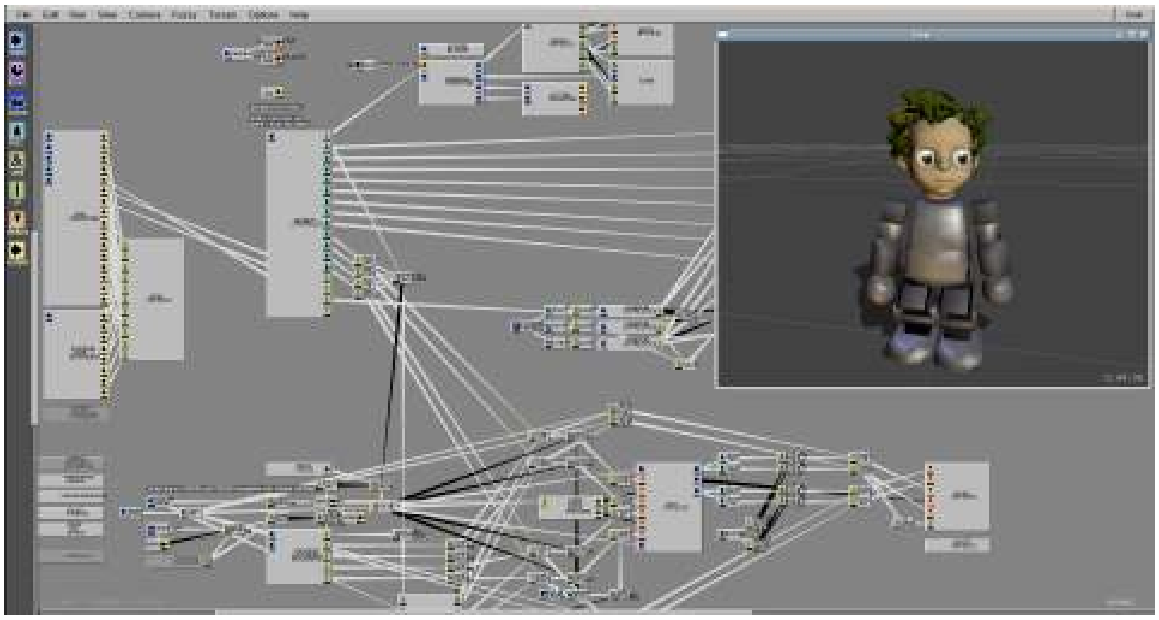
Instability

As development continued, our requirements changed. We had selected a key component for our robot control runtime architecture, a 3D animation system called Massive. Designed initially to bring artificial life to animation for crowds, Massive provides an agent centric behavior authoring environment and incorporates both procedural control through fuzzy logic and animation control based on key-framed or motion-capture data. However, at the time Massive was stable on Linux, not Windows. Using the Toolkit's streaming classes, we developed cross-platform streaming protocols for a robot control stream and a robot sensor stream (Listing 2a and 2b).

Customizing Massive

Massive has a plug-in architecture that can be customized. Each iteration produces changes in the fuzzy state of the fuzzy network for each agent. At each iteration, we can receive updates of the robot's movements, cognitive state, etc. However, the robot sensor data arrives at any time, therefore we spawned a thread to handle receiving sensor data asynchronously. The main application thread then picks up the sensor data as our plug-in is called on each iteration.

macros to using explicit user definable ids (e.g. ProtocolStream::motionframe_id in Listing 2b is simply an integer when streamed). The second problem was the way TCP was lagging as it waited briefly to attempt to aggregate additional writes to the socket (the Nagle algorithm). This resulted in multiple and/or partial motion frames in a single packet being sent from Massive to the rest of the robot control software. We needed to let the TCP layer know when an individual packet was ready to be sent so we implemented toggling the Nagle algorithm at the end of each iteration (Listing 3).



Massive screen showing biped robot under autonomous control. Using a custom robot plug-in the fuzzy network shown handles sensor data and determines the agent behavior that is then streamed to the actual robot.

Performance Issues

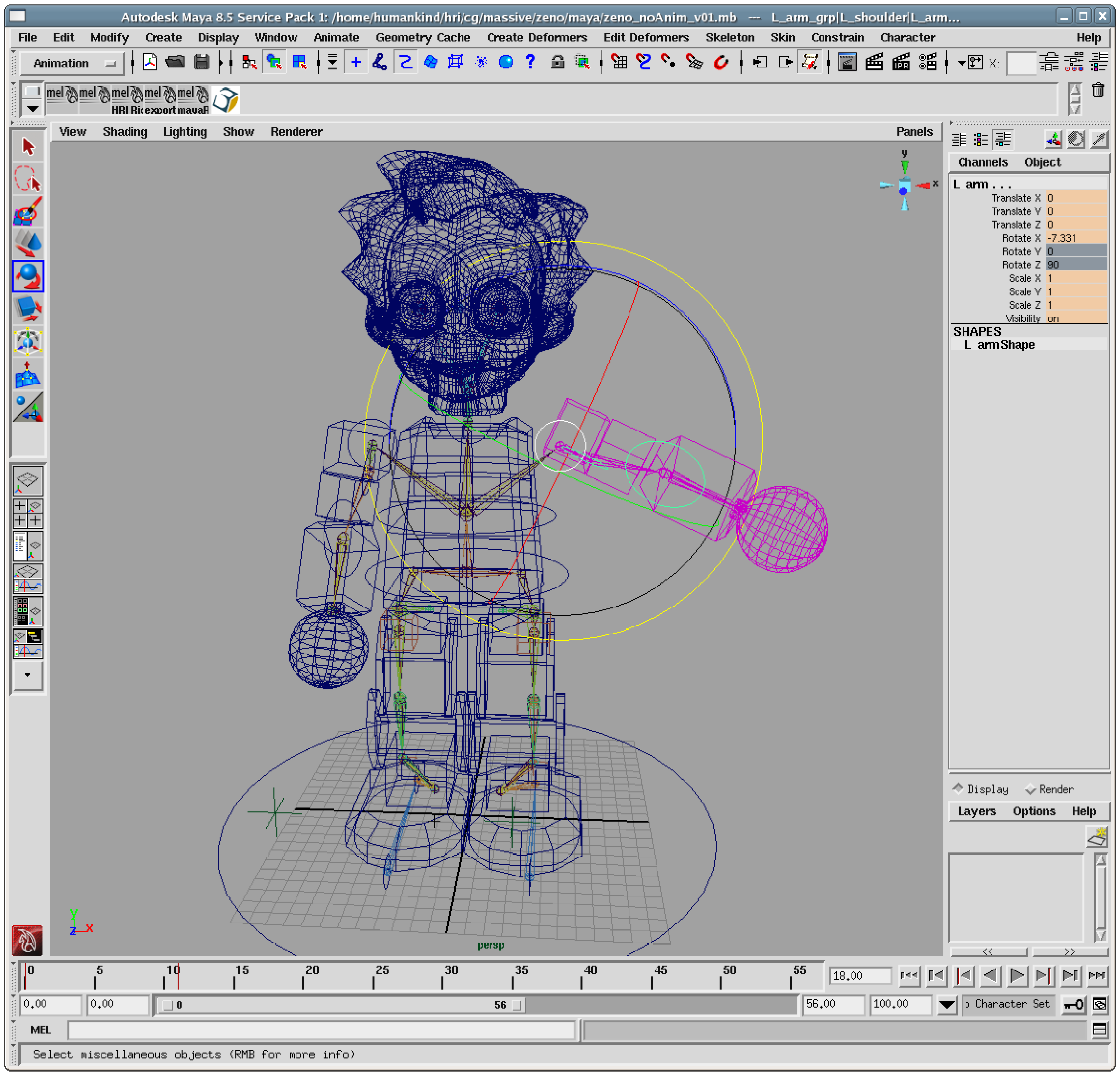
Initial problems included slow performance getting the control and sensor data to and from Massive in real-time. Packet analysis showed two problems. First, fully qualified class names for streamable classes were encoded as strings making the packets large. We switched from os_use_name_as_id in the streaming

Porting to Windows

When Massive became available on Windows we had another change to handle. Porting our custom robot plug-in was trivial except that the Nagle-toggling behavior did not produce the same steady packet stream on the Windows TCP stack. We went looking in our framework and found a useful class, os_cache, which follows the adaptor pattern. A simple change inserted the os_cache on top of the os_bstream which enabled our software to explicitly buffer and send each packet instead of trying to manipulate the TCP buffering/sending behavior. This allowed us to return to the steady stream of per-iteration packets (Listing 4).

Initial Results

With a functioning architecture in place we were ready to test biped robot animations. Autodesk's Maya software was the tool used to create the robot's model, rig, and animations. The animations were exported to Massive and tested. However, the prototype was not easy to animate successfully so another change in plans occurred. We decided to shortcut the process and test poses directly from Maya. We simply reused our basic C++ plug-in architecture as a native Maya plug-in. In Maya, the robot control plug-in sends the robot control stream but does not currently use the robot sensor stream.



Autodesk Maya 8.5 Service Pack 1: /home/humankind/hri/cg/massive/zeno/maya/zeno_noAnim_v01.mb --- L_arm_grp|L_shoulder|L_arm...

L arm...
Translate X 0
Translate Y 0
Translate Z 0
Rotate X -7.331
Rotate Y 0
Rotate Z 90
Scale X 1
Scale Y 1
Scale Z 1
Visibility on

SHAPES
L armShape

18.00
0.00 0.00 0 56 56.00 100.00

Select miscellaneous objects (RMB for more info)

Maya screen showing biped robot model and rig. Joint rotations can be sent direct to the robot via a MEL script and a native Maya plug-in.

Crawl, Walk, Run

Choosing highly flexible tools at the project initiation enabled us to easily respond to changes in project requirements. An adaptable C++ software framework from Recursion Software helped us meet core objectives in a complex software project. We stabilized thread performance and a clean shutdown, which were fundamental requirements.

As our requirements changed, and we wrote a native multithreaded plug-in to 3rd party software, in this case Massive. This entailed developing cross-platform streaming protocols for robot control and robot sensor data. When their software became stable on our preferred platform, we were able to migrate our plug-in with ease.

Because we were able to reduce packet size, we improved performance of our protocols. We also used available classes with appropriate design patterns to get a platform independent protocol over TCP that is suitable for soft real time control.

Finally, when requirements changed again, we were able to reuse most of our plug-in code to write a native plug-in for a different 3D animation package.
Many of the challenges we faced throughout the project are not uncommon, but predicting them with certainty is nearly impossible. The importance of due diligence when selecting third party tools that are commercially proven and provide a high level of supporting documentation and examples was key to our project's success.▪

To learn more about the products and resources used in this case study, visit the following websites:

Hanson Robotics:
www.hansonrobotics.com
Recursion Software, Inc.- C++ Toolkits:
www.recursionsw.com
Massive:
www.massivesoftware.com/robotics
Autodesk's Maya:
www.autodesk.com/maya

Listing 1 – Example wait for multiple objects

```
class RobotSensorHandler
{
private:
  // ...

  os_desc_t m_sensorDataEvent;      //
Thread wait object. For events
  os_desc_t m_terminateThreadEvent;  //
Thread wait object. For thread termination
};

RobotSensorHandler::RobotSensorHandler(
)
{
#ifdef OS_WIN32
  m_sensorDataEvent = CreateEvent(NULL,
false, false, NULL);           // Auto reset
  m_terminateThreadEvent =
CreateEvent(NULL, true, false, NULL);
// Manual reset
  if (m_sensorDataEvent == NULL ||
m_terminateThreadEvent == NULL)
  {
    throw std::runtime_error("CreateEvent()
failed");
  }
#else
  m_sensorDataEvent = 1; // For Linux
arbitrary but distinct values
  m_terminateThreadEvent = 2;
#endif
}

void RobotSensorHandler::streamProcess()
{
  os_handle_t waitHandles[ 2 ] =
  {
    os_ward::handle_for(
m_sensorDataEvent ),
    os_ward::handle_for(
m_terminateThreadEvent )
  };

  while ( os_ward::wait_for_multiple_objects(
2, waitHandles, false ) == 0 )
  {
    //
    // Process sensor events ...
    //
  }
}
```

Listing 2a – A sample streaming class header

```
#include <ospace/header.h>
#include <ospace/std/vector>

typedef double ServoPosition;
typedef std::pair<ServoId, ServoPosition>
ServoMove;

class MotionFrame;

// Declare streaming functions.
void os_write( os_bstream& stream, const
MotionFrame& object );
void os_read( os_bstream& stream,
MotionFrame& object );

class MotionFrame
{
public:

  friend void os_write( os_bstream& stream,
const MotionFrame& object );
  friend void os_read( os_bstream& stream,
MotionFrame& object );

  typedef std::vector<ServoMove> MoveList;

  // ...

private:
  int m_duration;
  MoveList m_moves;
};

OS_CLASS( MotionFrame );
OS_STREAM_OPERATORS( MotionFrame
);
```

Listing 2b – A sample streaming class

```
#include <ospace/source.h>
#include <ospace/std/vector>
#include <ospace/std/iostream>
#include <ospace/stream.h>
#include <ospace/uss/std/vector.h>
#include <ospace/uss/std/pair.h>
#include "ProtocolStream.h"
#include "MotionFrame.h"

#if defined OS_USE_ALTERNATE_STD ||
defined OS_OSPACE_STD_NAMESPACE
  using OS_STD os_pair;
  using OS_STD os_string;
  using OS_STD os_vector;
#endif

using ProtocolStream;

// ServoMove is based on std::pair ... the
STL streamer functions are in the
// default namespace. MoveList is a
std::vector

// We must have a binary stream macro for
each parameterized STL type we use
OS_STREAMABLE_0( (ServoMove*),
ProtocolStream::servomove_id );

OS_STREAMABLE_0(
(MotionFrame::MoveList*),
ProtocolStream::movelist_id );

OS_STREAMABLE_0( (MotionFrame*),
ProtocolStream::motionframe_id );

// We have to author the MotionFrame
streaming methods.

void os_write( os_bstream& stream, const
MotionFrame& object )
{
  stream << object.m_duration <<
object.m_moves;
}
```

```
void os_read( os_bstream& stream,
MotionFrame& object )
{
  stream >> object.m_duration >>
object.m_moves;
}

// ...
```

Listing 3 – Implementation 1. Writes to the stream are to the underlying socket. Toggle the Nagle behavior to flush out a packet.

```
RobotControlClient::RobotControlClient(con
st os_socket_address & address)
{
  m_socket.connect_to( m_address );

  // Create a binary stream on the socket.
  this->m_stream = new os_bstream(
m_socket );
}


void RobotControlClient::streamFlush()
{
  int flag = 1;
  m_socket.setsockopt( IPPROTO_TCP,
                       TCP_NODELAY,   // Disable
Nagle

reinterpret_cast<char*>(&flag),
                       sizeof(int) );

  flag = 0;
  m_socket.setsockopt( IPPROTO_TCP,
                       TCP_NODELAY,   // Re-
enable Nagle

reinterpret_cast<char*>(&flag),
                       sizeof(int) );
}
```

Listing 4 – Implementation 2. Using os_cache to buffer writes to the stream until the cache is flushed.

```
RobotControlClient::RobotControlClient(con
st os_socket_address & address)
{
  m_socket.connect_to( m_address );
  int flag = 1;
  m_socket.setsockopt( IPPROTO_TCP,
                       TCP_NODELAY,   // Disable
Nagle

reinterpret_cast<char*>(&flag),
                       sizeof(int) );


  this->m_cache = new os_cache (
m_socket, 16384, 0);

  // Create a binary stream on the socket.
  this->m_stream = new os_bstream(
*m_cache );
}


void RobotControlClient::streamFlush()
{
  this->m_stream->adapter().device()-
>sync();
}
```

About the Author

Bob Hauser, Director of Software, Hanson Robotics, Inc.
With over 15 years in software engineering, Bob Hauser has an extensive background in C, C++, C#, and Java, and developing natural language processing work in semantics. Mr. Hauser has nine patents in distributed computing and has held positions at Recursion Software and Language Computer. He has a BS from the University of Northern Iowa and an MS from Michigan State University, where he specialized in artificial intelligence.

RECURSION SOFTWARE, Inc.

About Recursion Software, Inc.

Recursion Software is an innovative provider of intelligent middleware and distributed computing solutions based on Service Oriented Architecture (SOA) principles and interoperability standards between multiple languages and platforms. Recursion products help enterprises to extend their current application architecture while providing the tools developers need to build the next generation of intelligent, mobile, applications. The company is a small, privately held corporation, located in the Dallas-Fort Worth area with a large customer base of government and commercial clients across the world. Since 1993, our products have enabled complex, performance-oriented software development solutions for mission-critical applications and systems. The majority of clients are in the defense, financial, energy, computer technology, and telecommunications industries.

Recursion Software is regarded for its Voyager Edge platform, a powerful agent-based interoperable platform that supports a total range of edge devices, including handheld devices, PDAs, sensors and cameras. The company remains the leading proponent and preferred platform for intelligent mobile agent and agent space technology and has been issued more than 18 patents related to distributed computing, with 30 patents in various states of pending and filing.