Multiplayer Game Programming

Lecture 8

ITP 484

Review!

- Explain how to reduce the bits needed to serialize a 3 component vector
- Explain how to reduce the bits needed to serialize a quaternion
- Explain how to optimize the number of bits needed to represent a 10 bit number when you know that one particular value is 100 times more common than the others

Lab comments

- Timer considerations- bad frame rate, frame spikes. Subtract but not too much!
- Don't make small allocations on heap for packet char code!
- Allocate in constructor, deallocate in destructor!!
- Don't mix Malloc and delete!
- Process more than one packet per frame! Why?
- Don't unnecessarily couple your gameplay code to your network code! Checking object type DOES NOT BELONG IN NETWORK CODE!

Remote Procedure Calls

- Remote Procedure Call
  - Invocation of remote procedure
  - Details of network layer abstracted away
  - Calling requires identifier, parameters and RPC API
- Remote Method Invocation
  - Object Oriented version of RPC
  - Requires additional object as target of method

Supported Parameter Types

- What can you serialize easily:
  - Primitives
    - uint_#t, int_#t, float, double, char
  - Structs of Primitives (PODs)
  - Arrays of PODs or Primitives
- With more work:
  - Dynamically allocated objects
    - Send each object with an id first
    - Then refer to the objects by id
    - `bool IsSameObject( Obj* inA, Obj* inB )`

Return Values

- How does callee send results to caller?
  - It doesn’t!
  - Predetermined RPC to call in response
  - Caller passes RPC identifier as param
- When does caller get results?
  - RPC API can block until response
  - RPC API can allow run loop polling for response
  - RPC API can call response on thread

Reliability

- RPCs can be reliable or unreliable
- Depends on API and underlying transport protocol
  - Reliability built on top of unreliable transport
  - Reliability granted by reliable transport
- Notification of RPC execution can be handled in RPC Module or at a higher level

Example APIs

- Naïve Scripting Language
- Stub/proxy
  - ONC-RPC
- HTTP
  - XML-RPC
  - JSON-RPC
  - REST

Naïve Scripting Language

- Serialize text of the call as a string
- Send it over the network
- Evaluate it on the other end
- Send a response back as a string
- Limitations?
    - Only POD parameters
    - Requires scripting language
    - Not space efficient
    - No type checking on client

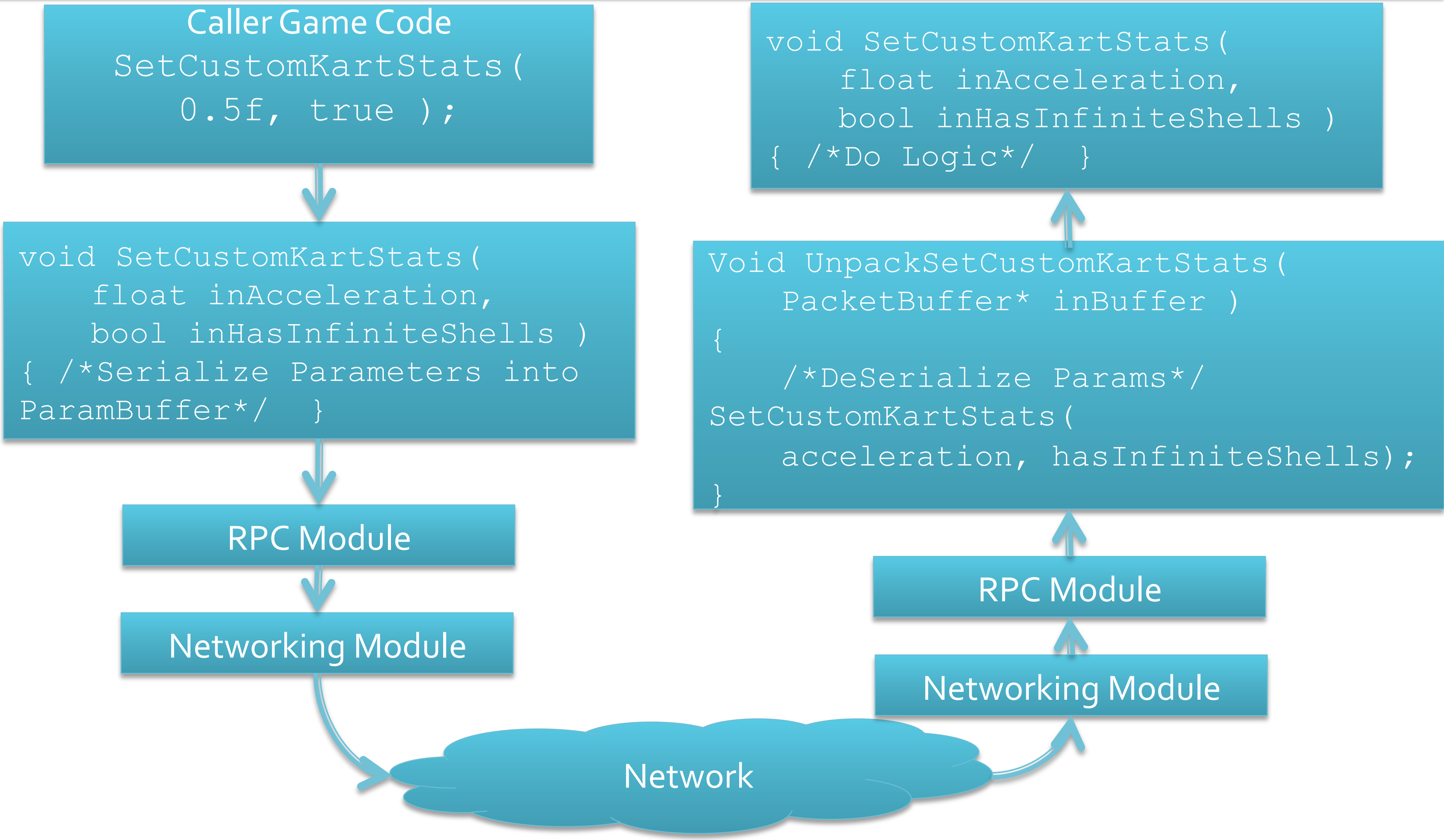
Stub/Proxy API

- RPCs look just like regular function calls
- Caller RPC module packages up function and parameters
- Sends to callee over network
- Callee RPC module unpackages functions and parameters
- Callee executes function

Example

```
void SetCustomKartStats(
    float inAcceleration,
    bool inHasInfiniteShells )
{
    gKart->mAcceleration =
        inAcceleration;
    gKart->mHasInfinitShells =
        inInfiniteShells;
}
```

Flow



Caller Game Code

```
SetCustomKartStats(
0.5f, true );
```

```
void SetCustomKartStats(
    float inAcceleration,
    bool inHasInfiniteShells )
{ /*Serialize Parameters into
ParamBuffer*/  }
```

RPC Module

Networking Module

Network

Networking Module

RPC Module

```
Void UnpackSetCustomKartStats(
    PacketBuffer* inBuffer )
{
    /*DeSerialize Params*/
SetCustomKartStats(
    acceleration, hasInfiniteShells);
}
```

```
void SetCustomKartStats(
    float inAcceleration,
    bool inHasInfiniteShells )
{ /*Do Logic*/  }
```

Example Caller Stub code

```
void SetCustomKartStats(
     float inAcceleration,
     bool inHasInfiniteShells )
{
     MemoryOutputStream mos;

     mos.WriteString(
           “SetCustomKartStats” );
     mos.WriteData(
           &inAcceleration, sizeof( float ) );
     mos.WriteBits(
           &inHasInfiniteShells, 1 );

     RPCModule::sInstance->BatchRPC( mos );
}
```

Example Callee Stub Code

```
void UnpackSetCustomKartStats(
     MemoryInputStream* inMIS)
{
     float acceleration;
     inMIS->ReadData(
          &acceleration, sizeof( float ) );
     uint8_t hasInfiniteShells;
     inMIS->ReadData(
          &hasInfiniteShells, 1 );
     SetCustomKartStats(
          acceleration,
          hasInfiniteShells != 0 );
}
```

Stub Generation

- Usually Automatic
  - Function definition in tool specific language
    - RPCL
    - IDL
    - Any reflect-able language!
  - Tool outputs stub code, compiled into process

Example of RPCL

```
struct SetCustomKartStats_call
{
       float        inAcceleration;
       boolean      inHasInfiniteShells;
};

program TrojarioKart
{
       version TrojanKartVersion1
       {
             void SetCustomKartStats(
                    SetCustomKartStats_call ) = 1;
       } = 1;
} = 0x2e248452
```

How does RPC Module know which unwrap function to call?

```
typedef void ( *UnpackRPCProc )( MemoryInputStream*
inMIS );

std::unordered_map< std::string, UnpackRPCProc > gRPCs;

gRPCs[ “SetCustomKartStats” ] = UnpackSetCustomKartStats;

void RPCModule::HandleIncomingRPC(
      MemoryInputStream* inMIS )
{
      string procName;
      inMIS->ReadString( procName );
      UnpackRPCProc proc = gRPCs[procName ];
      proc( inMIS );
}
```

RPC Function Map

- Could be auto filled in when stubs are generated
- Could use more efficient type than strings
  - ONC-RPC uses integers

HTTP

- Very popular- RPC Calls transferred as text
- Examples
  - XML-RPC
  - JSON-RPC
  - REST
- Advantages
  - Easy to understand
  - Easy to debug
  - Discoverability
- Disadvantages
  - Bandwidth Inefficient
  - Not type safe!

XML-RPC

```
<?xml version=“1.0”?>

<methodCall>
        <methodName>TrojarioKart.SetCustomKartStats</methodName>
        <params>
                <param><value><float>0.5</float></value></param>
                <param><value><boolean>1</boolean></value></param>
        </params>
</methodCall>
```

JSON-RPC

```
{
        "method": "SetCustomKartStats",
        "params":
        [
                0.5,
                1
        ],
        "id": 1
}
```

JSON-RPC response

```
{
        “result” :
        {
        },
        “error” :
        {
                msg: “Too much acceleration!”
        },
        “id”: 1
}
```

Unique id must match request

REST

- Representational State Transfer
- Current Favorite
- Requires least intermediate infrastructure
- More human readable than JSON-RPC
- More bandwidth efficient than XML-RPC
- HTTP already handles RPCs!

REST examples

- GET
  - http://MyServer.com/TrojarioKart?methodName=SetCustomKartStats&acceleration=0.5&hasInfiniteShells=1
  - http://MyServer.com/TrojarioKart/SetCustomKartStats?acceleration=0.5&hasInfiniteShells=1
  - http://TrojarioKart.com/SetCustomKartStats?acceleration=0.5&hasInfiniteShells=1

REST examples

- POST
  - http://TrojarioKart.com/SetCustomKartStats
    - In body, send form data

      acceleration=0.5&hasInfiniteShells=1
  - Better for sending large amounts of data
    - Binary data can be base 64 encoded
  - Worse for testing from web browser

API wrapping

- Programming API does not have to match information protocol
- Build stubs on caller that use HTTP / XML-RPC to talk to server
- Isolates type safety problem into API

Example

```
void SetCustomKartStats(
     float inAcceleration,
     bool inHasInfiniteShells )
{
     char buff[ 512 ];
     sprintf( buff, 512,
http://TrojarioKart.com/
SetCustomKartStats/?acceleration=
%f&hasInfiniteShells=%d,
     inAcceleration, inHasInfiniteShells );
     HTTPModule::sInstance->Get( buff );
}
```