Object Replication & Serialization

Chapter 3-5

GAME 311- Network Programming

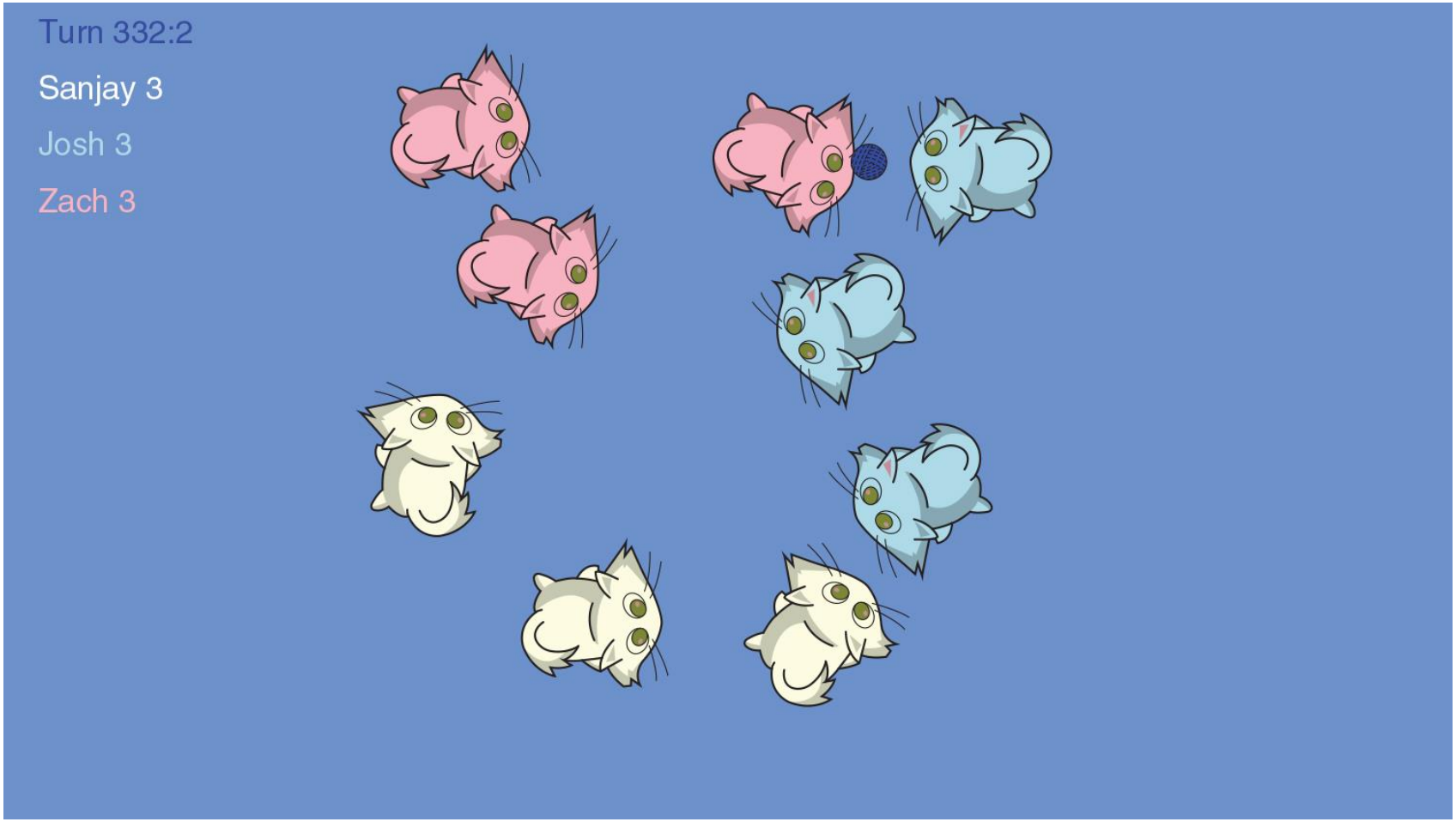
Chapter 3 & 4 Objectives

- **Serialization**
  - Why can't we just memcpy everything between hosts?
  - Streams, endianness, and bits
- **Referenced data**
  - What happens when data points to other data?
- **Compression**
  - How to serialize efficiently
- **Maintainability**
  - Bug-resistant serialization
  - Data-driven serialization

Replicating Objects

- If we send each piece of data off one by one for the X, Y and paddle info like colour and size we could get offset in communication.
- All of a sudden we start accidentally mixing packets intended for the X location as the Y and data mixed up.
- The better option is to replicate entire objects at a time.
- Send all the information about one particular object as required.

Case Study (RTS)



Replicating Objects

- Consider the following class of `RoboCat`

```
class RoboCat
{
public:
    RoboCat(){}
private:
    int mHealth;
    int mMeowCount;
};
```

- How can we send an instance to a remote host?

Send Code Review:

- Our send call is directly sending a char* (series of bytes)

```
// BOTH
void NetworkManager::SendData(const char* data)
{
    int messageLen = MAX_MESSAGE_SIZE;

    if (send(peerSocket, data, messageLen, 0) == SOCKET_ERROR)
    {
        int error = WSAGetLastError();

        if (error != WSAEWOULDBLOCK)
        {
            cout << "[Error] send error: " << WSAGetLastError() << endl;
            Shutdown();
        }
    }
}
```

Naïve Approach to Replication

```
void NaivelySendRoboCat( int inSocket, const RoboCat* inRoboCat)
{
    send( nSocket,
        reinterpret_cast< const char* >( inRoboCat ),
        sizeof( RoboCat ), 0 );
}

void NaivelyReceiveRoboCat( int inSocket, RoboCat* outRoboCat)
{
    recv( inSocket,
          reinterpret_cast< char* >( outRoboCat ),
          sizeof( RoboCat ), 0 );
}
```

- Will this work?
  - We can reinterpret_cast our entire objects memory footprint into a char*.

What About This RoboCat?

```
class RoboCat : public GameObject
{
public:
    RoboCat()
    {
        mName[ 0 ] = '\0';
    }
    virtual void Update();

private:
    int32_t                 mHealth;
    int32_t                 mMeowCount;
    GameObject*             mHomeBase;
    char                    mName[ 128 ];
    std::vector< int32_t >  mMiceIndices;
    Vector3                 mLocation;
    Quaternion              mRotation;
};
```

Problems for Advanced RoboCat

- **mHomeBase** is a pointer:
  - Copying the value of a pointer doesn't copy the data.
- **mMiceIndices** is a complex data type:
  - Calling **memcpy** on the insides might not copy the data.
  - The vector on the remote host must be initialized to the proper size.

```
class RoboCat : public GameObject
{
public:
    RoboCat()
    {
        mName[ 0 ] = '\0';
    }
    virtual void Update();

private:
    int32_t                     mHealth;
    int32_t                     mMeowCou
    GameObject*                 mHomeBas
    char                        mName[12
    std::vector< int32_t >      mMiceInd
    Vector3                     mLocatio
    Quaternion                  mRotatio
};
```

Problems for Advanced RoboCat

The virtual `Update()` requires a virtual function table pointer:

- **Copying the value of the function table pointer is pointless.**

`mName` is 128 bytes, unlikely all are used:

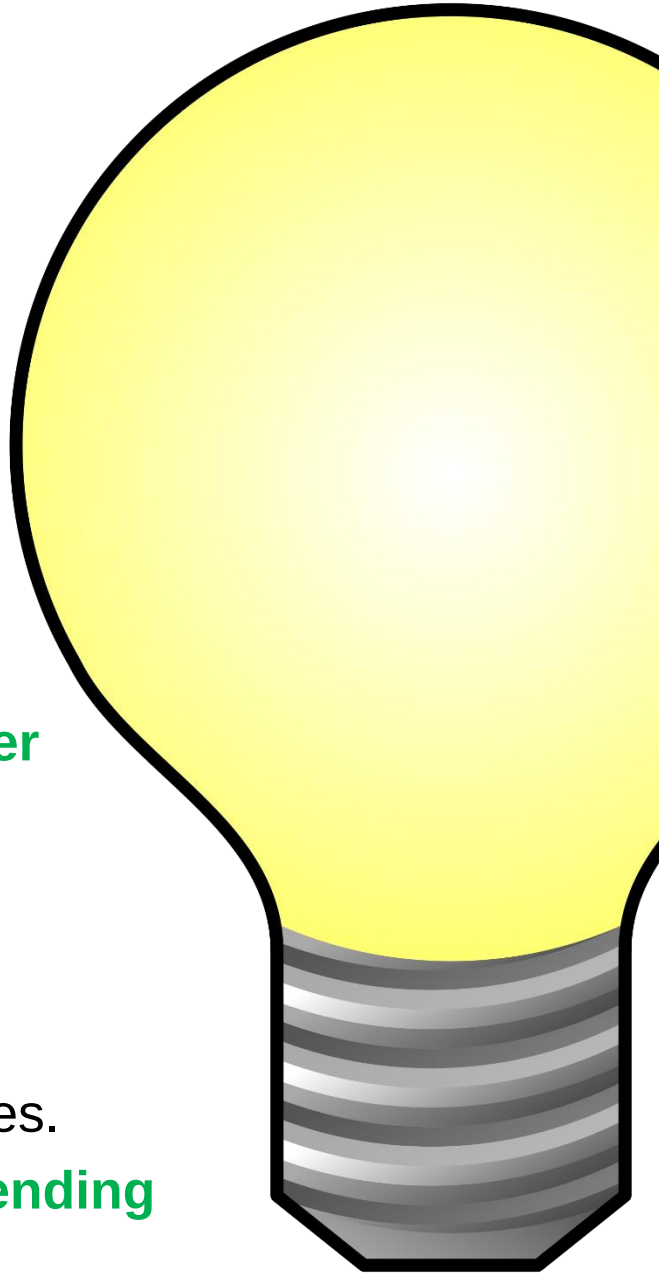
- `memcpy` will copy all 128 bytes regardless.

```
class RoboCat : public GameObject
{
public:
    RoboCat()
    {
        mName[ 0 ] = '\0';
    }
    virtual void Update();

private:
    int32_t                     mHealth;
    int32_t                     mMeowCount
    GameObject*                 mHomeBase;
    char                        mName[128]
    std::vector< int32_t >      mMiceIndic
    Vector3                     mLocation;
    Quaternion                  mRotation;
};
```

Solution

- Replicate each member variable **individually**:
  - Replicating **pointers:**
    - **Replicate the data they point to**.
  - Replicating **vectors:**
    - **replicate their size and data.**
  - Replicating objects with **virtual methods**:
    - **Replicate identifiers to hook up proper virtual function tables.**
  - Replicating **large arrays**:
    - **omit unused elements.**
  - But packets should be as big as possible:
    - Replicated variables usually < 1300 bytes.
    - **Combine member variables before sending them by serializing into a buffer**.



Custom Read and Write

```
void RoboCat::Write( OutputMemoryStream& outStream ) const
{
    outStream.Write( mHealth );
    outStream.Write( mMeowCount );
    //no solution for mHomeBase yet
    outStream.Write( mName, sizeof( mName ) );
    //no solution for mMiceIndices
    outStream.Write( mLocation, sizeof( mLocation ) );
    outStream.Write( mRotation, sizeof( mRotation ) );
}

void RoboCat::Read( InputMemoryStream& inStream )
{
    inStream.Read( mHealth );
    inStream.Read( mMeowCount );
    //no solution for mHomeBase yet
    inStream.Read( mName, sizeof( mName ) );
    //no solution for mMiceIndices
    inStream.Read( mLocation, sizeof( mLocation ) );
    inStream.Read( mRotation, sizeof( mRotation ) );
}
```

Putting It All Together

```
void SendRoboCat( int inSocket, const RoboCat* inRoboCat )
{
    OutputMemoryStream stream;
    inRoboCat->Write( stream );
    send( inSocket, stream.GetBufferPtr(),
          stream.GetLength(), 0 );
}

void ReceiveRoboCat( int inSocket, RoboCat* outRoboCat )
{
    char* temporaryBuffer =
       static_cast< char* >( std::malloc( kMaxPacketSize ) );
    int receivedByteCount =
       recv( inSocket, temporaryBuffer, kMaxPacketSize, 0 );

    if( receivedByteCount > 0 )
    {
        InputMemoryStream stream( temporaryBuffer,
          static_cast< uint32_t > ( receivedByteCount ) );
        outRoboCat->Read( stream );
    } else {
        std::free( temporaryBuffer );
    }
}
```

Maintainability

- `Read` and `Write` for data type must remain in sync with each other:
  - Changing the order or compression in one method requires changing the other.
  - Opportunity for bugs.
- `Read` and `Write` must remain in sync with data type!
  - Adding or removing member variable
  - Changing data type of member variable
    - Including changing precision required
  - All opportunities for bugs
- **Tradeoff**: Improving maintainability can mean decreasing performance.

Referenced Data

- Pointer member variables **reference** other data.
- **Container** member variables, like **`vector`**, also reference other data.
- Data can be referenced by only one object, or by multiple objects:

```
class RoboCat : public GameObject
{
public:
    RoboCat()
    {
        mName[ 0 ] = '\0';
    }
    virtual void Update();

private:
    int32_t                 mHealth;
    int32_t                 mMeowCount
    GameObject*             mHomeBase;
    char                    mName[128]
    std::vector< int32_t >  mMiceIndic
    Vector3                 mLocation;
    Quaternion              mRotation;
};
```

Problems for Advanced RoboCat

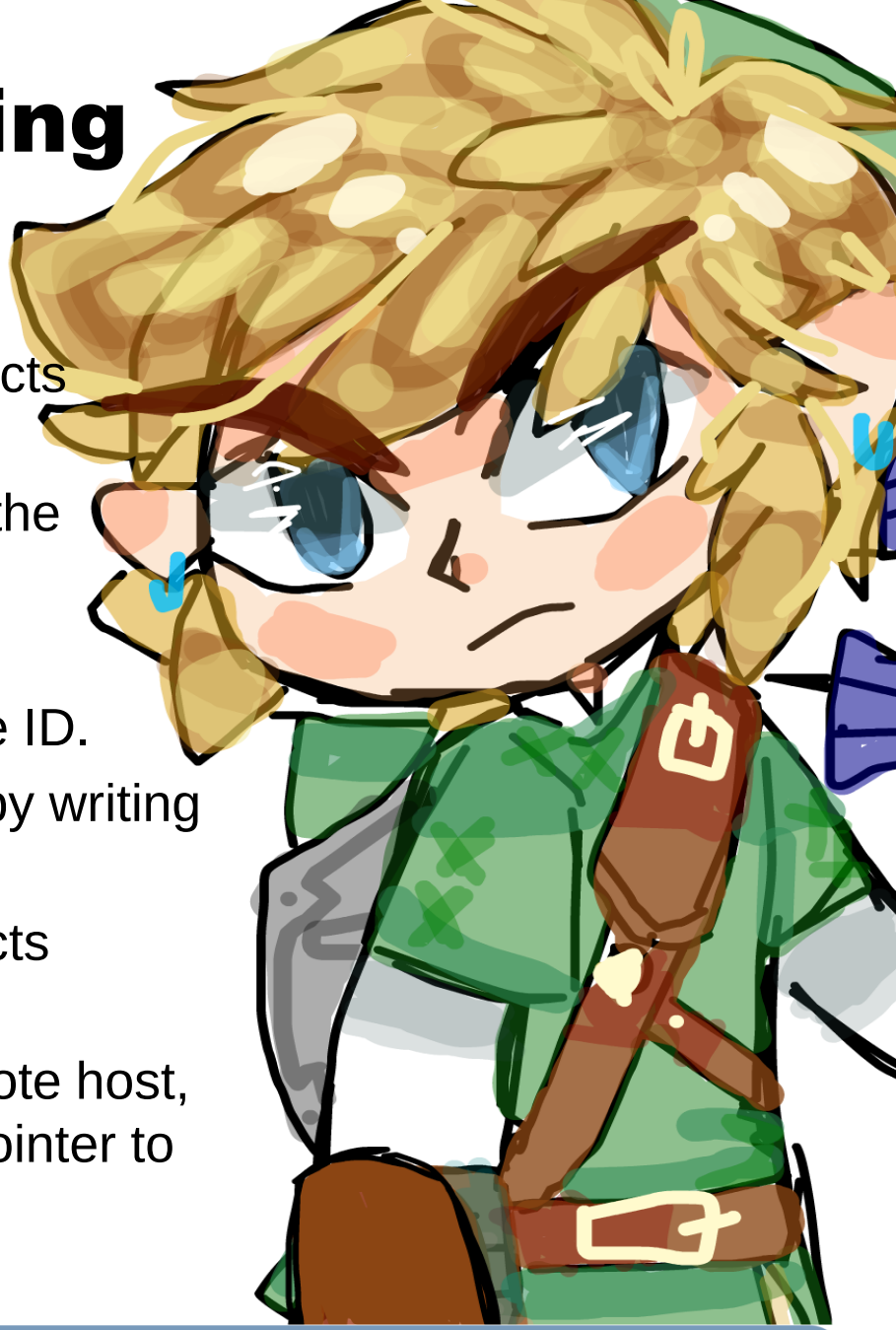
If multiple `RoboCat`s have the same `HomeBase`, that `HomeBase` is referenced by multiple objects

Each `RoboCat` has a unique vector of mice indices it is tracking; each vector is referenced/owned by only a single `RoboCat`.

```
class RoboCat : public GameObject
{
public:
    RoboCat()
    {
        mName[ 0 ] = '\0';
    }
    virtual void Update();

private:
    int32_t                     mHealth;
    int32_t                     mMeowCount
    GameObject*                 mHomeBase;
    char                        mName[128]
    std::vector< int32_t >      mMiceIndic
    Vector3                     mLocation;
    Quaternion                  mRotation;
};
```

Linking



- Data referenced by multiple objects should not be embedded:
  - Results in multiple copies of the object when deserialized.
- Use linking:
  - Assign object needs a unique ID.
  - Serialize pointers to objects by writing the object's ID:
    - Serialize referenced objects separately/earlier.
  - When deserializing on a remote host, replace an object ID with a pointer to a deserialized object.

Fixed Point

- Games use **floating-point** numbers for most math.
- Floating point numbers are 32 bits.
- Values often don't require 32 bits of precision.
  - Leave as floats for fast calculation at run time.
  - Convert to smaller precision **fixed-point** numbers during serialization.

- Fixed-point numbers are integers with an implied constant division.

Fixed-Point Example

RobotCat:mLocation

- World extends from –2000 games units to 2000 game units.
- mX and mZ position values only need to be accurate to within 0.1 game units.
- How many possible relevant values are there for mX then?

  (MaxValue – MinValue) / Precision + 1

  (2000 – –2000) / 0.1 + 1 = 40001

```
class RoboCat : public GameObject
{
public:
    RoboCat()
    {
        mName[ 0 ] = '\0';
    }
    virtual void Update();

private:
    int32_t                     mHealt
    int32_t                     mMeowC
    GameObject*                 mHomeB
    char                        mName[
    std::vector< int32_t >      mMiceI
    Vector3                     mLocat
    Quaternion                  mRotat
};
```

Fixed-Point Example

RobotCat:mLocation

(MaxValue – MinValue) / Precision + 1

(2000 – –2000) / 0.1 + 1 = 40001

- There are only 40,001 possible values for mX.
  - All values representable with only 16 bits.

```
class RoboCat : public GameObject
{
public:
    RoboCat()
    {
        mName[ 0 ] = '\0';
    }
    virtual void Update();

private:
    int32_t                     mHealt
    int32_t                     mMeowC
    GameObject*                 mHomeB
    char                        mName[
    std::vector< int32_t >      mMiceI
    Vector3                     mLocat
    Quaternion                  mRotat
};
```

Bit Streams

- Not all data requires a multiple of 8 bits:
  - For example, a boolean value can be stored in a single bit!
- Bandwidth is expensive: Use as few bits as possible.
- Current streams support only byte-level precision.
- Bit Streams
  - Manager which takes in a variable and adds only the actual bits used to the stream.
  - Each time we write a value into a stream, we should be able to specify bit count.
  - Consider writing 13 as a 5-bit number and then 52 as a 6-bit number.

Chapter 3 & 4 Summary

- Explain the issues with floating point numbers
- Serialize an object with pointers
- Efficiently compress data for transmission.
- Explain the important data transmitted with an object for replication