Platitude: Software Design Description

CS121 - Software Development | Fall 2010

Harvey Mudd College ¶

Project team: Philip Aelion-Moss, Christopher Beavers, David Cook, Sarah Ferraro, Aaron Gable, Leif Gaebler, Jacob Heller, My Ho, Sean Laguna, Kathryn Lingel, Camille Marvin, Stephanos Matsumoto, Beatrice Metitiri, Emily Myers-Stanhope, Oliver Ortlieb, Thea Osinski, Richard Porczak, Kevin Riley, Max Strater, Rahul Swaminathan, Russel Transue, Heather Williams, Lilian de Greef, Alexandre François (pm).

Table of Contents

- System overview
- Conventions
- Common elements
  - The Game State
  - Platformer Player
  - Builder Player
  - Items
- Modules
  - Graphics Modules
  - Music Module
  - Networking Module
  - Physics Module
  - Control Modules
  - Gameplay Module
- Appendix A - Getting started with the code
- Appendix B - Development plan and schedule
  - Schedule
  - Milestones
    - Prototype 1: basics
    - Prototype 2: builder and platformer
    - Prototype 3: clients and server
    - Demo system

System Overview

Conventions

Game name: Platitude namespace ptd (can be interpreted as Platitude, or Platformer Tower Defense!)

Common Elements

- the game state: a class containing all the elements of the game (platformer players, builder player, level map, etc.)

The Game State

Game State Team: Camille Marvin, Emily Myers-Stanhope, Jacob Heller

`ptd::CGameState` is the top level container (and public interface) for all game elements stored in the main repository. `ptd::CGameState` is derived from `fsf::CNode`. It is defined in the Game module (`PtdGameModule.[h|cc]`).

Elements in the game state:

- Note: This is the somewhat high level description of the objects in the game state. For more detailed descriptions of these objects (CCollisionObjects, etc), read the short paragraphs on them in the Physics Module section of the document. For descriptions of these new classes that are closer to the code, read the section devoted to them later just after the Game State section of the document.
- std::list<CCollisionObject*> m_collisionObjects - a list of all of the objects in the game state that are not affected by physics and aren't part of the original level map. In the demo, this only contains objects added by the builder during play
- std::list<CMobileObject*> m_mobileObjects - a list of all the objects in the game state that are affected by physics, with the exception of the platformers. This would theoretically contain monsters and moving platforms (added by the builder or part of the original map), but moving obstacles were not part of the demo, so this list remained empty.
- std::list<CCollisionObject*> m_level - contains all of the platforms that make up the level. These are added to the game state when it is constructed, and cannot be destroyed.
- Object IDs
  - Object IDs are used to uniquely identify each object in game state. When an object is added to the game state, it is given an object ID. The module that added the object can then use this object ID to interact with this specific object, rather than trawling through one of the lists to find it each time it needs to alter something.
  - m_nextObjectID - keeps track of the next new value that can be assigned as an object ID. Starts at NUM_PLAYERS (the number of players allowed in the game) and goes up from there.
  - m_availableObjectIDs - a "recycling" list for old object IDs. This list starts out empty, and as objects are removed from the game state, their object IDs are added to this list.
- Map size
  - m_mapHeight
  - m_mapWidth
- Platformers
  - std::list<CPlatformer*> m_platformers - list of players in the game.
  - std::list<int> m_availablePlayerIDs - list of player IDs not yet taken. The list is initialized to contain playerIDs 1 through NUM_PLAYERS where NUM_PLAYERS is the maximum number of players allowed in the game. When a player is added it is given an ID, which is removed from the list. When a player disconnects, it's ID is added back to this list. When the list is empty, new players who try to connect will get an error message telling them that the game is already full.
- Builder
  - CBuilder* m_builder - the builder (and their inventory)
  - m_itemPressed - keeps track of the item most recently pressed by the builder on their screen. Helps with the transition of the item from being an inventory item to being an obstacle in the game.
- Score (not used in demo)
  - int m_score
- Various Mutexes
  - everything in the game state has a mutex for multithreading purposes

Accessing Data members in the Game State : The game state contains all elements that represent the state of the game. All other modules revolve around the data stored here.

- The game state contains four lists: (1) level pieces, (2) builder pieces (these are called "collision objects" in the game state), (3) mobile objects, and (4) player objects. A pointer directly to any of these lists would violate the encapsulation policy.
  - There are methods for each list that return begin and end iterators for reading (and occasionally manipulating) all of the items in a given list.
    - getIterCollisionObj, getIterCollisionObjEnd
    - getIterMobileObjects, getIterMobileObjectsEnd

        - getIterLevelPieces, getIterCollisionObjEnd
        - getIterCollisionObj, getIterCollisionObjEnd
    - There are also methods for accessing and manipulating specific elements of these lists using their object ID.
        - CPlatformer* getPlatformer(int PlayerID)
        - CCollisionObject* getCollisionObject(int objectID)
        - CMobileObject* getMobileObject(int objectID)
    - Every object type in these lists contains a *lock()* method, which must be called before mutating that object (and the corresponding *unlock()* method should be called when done).
    - A list may be mutated through one of the *add()* or *remove()* methods which correspond to that list. The add methods return object IDs, and the remove methods require them.
        - int addCollisionObject(CCollisionObject* object); void removeCollisionObject(int objectID)
        - etc...(replace CollisionObject with MobileObject, etc)
    - The lists themselves have a *lock()* method as well, but these should only be used when the entire list is mutated (e.g. adding or removing an element).

Other Game State Functions :

- The game state contains an `initialize()` function that loads floats representing the coordinates of every level piece from the file `map.txt` and populates the LevelPieces list with rectangular blocks.

Classes

Several new classes needed to be created for this game. They are mostly described in their individual sections below, but they have some functions in common that are described here.

- Getters and Setters
    - Nearly all of the private data members have public getters and setters. Where the name of a variable is variableName and its type is variable_type, they take the following form:
        - variable_type getVariableName() const
        - void setVariableName(variable_type variableNameOrSomething)
- Networking Functions
    - int bufferSize() - the size, in bytes, that this object will need when converted to a character buffer
    - char* toBuffer(char* pData) - encodes the object as a character buffer.
    - char* fromBuffer(char* pData) - decodes a character buffer and sets all of the data members of the object.

Platformer Player

- Class CPlatformer
    - Inherits all data members and functions from CMobileObject. Additional data members are described below
    - Limiting factors:
        - double m_maximumVelocity
        - double m_maximumAcceleration
        - double m_jumpHeight
    - Current state of platformer:
        - bool m_moveLeft - is the platformer moving left?
        - bool m_moveRight
        - bool m_moveUp
        - bool m_moveDown
        - bool m_jump

        - bool m_faceRight
        - bool m_cling
    - Members needed for climbing
        - CCollisionObject* m_clingLeft
        - CCollisionObject* m_clingRight
    - bool m_humanControlled - human or AI player? (our demo has no AI, so this m_humanControlled is set to true by default)
    - int m_playerID - for distinguishing platformers
    - fsf::CMutex m_csPlatformer

Builder Player

- Class CBuilder
    - bool m_humanControlled - human or AI player? (our demo has no AI, so this m_humanControlled is set to true by default)
    - int m_money - money available to the builder to buy items from the inventory. If m_money is less than the price of an object, the builder won't be able to add it the game state.
    - std::list<CBuilderObject*> m_inventory - an list of all the types of obstacles available to the builder. It's more of a store than an inventory, as it theoretically contains inexhaustible amounts of each obstacle and the builder cannot use an obstacle until s/he has paid for it.

- Class CBuilderObject
    - This class inherits from CCollisionObject and is used for the items in the builder's inventory. Objects of this class are basically Collision Objects, but they also have a price and a name to display to the builder. When the builder selects one of these from the inventory, it is added to the m_collisionObjects list of the Game State and loses these extra attributes.
    - Inherits all data members and functions from CCollisionObject
    - double m_cost - the cost to the builder to place an object of this type in the game
    - std::string m_label - A string describing the item to the builder (not used in demo)

Items and Obstacles

- Class CCollisionObject
    - An object that cannot be affected by physics. At the time of the demo, the builder could only add collision objects to the game state.
    - Position and Size
        - double m_xCoord
        - double m_yCoord
        - double m_width
        - double m_height
    - Physics related - Even though Collision Objects are not themselves affected by physics (other than collision detection, hence the name), they can affect other objects.
        - double m_friction - coefficient of friction of the object. Affects mobile objects and platformers in contact with the object.
        - bool m_isClimbable - true if the object is climbable. By default this is true, but to make things more challenging for the platformer it can be switched to false.
    - double m_color[3] - Color of the object in RGB
    - int m_objectID - unique identifier for the object. It is initialized to 0, then set if/when it is added to the game state.
    - fsf::CMutex m_csCollisionObject - requisite mutex.

- Class CMobileObject
    - The class for objects that can be affected by physics
    - Inherits all data members and functions from CCollisionObject. Additional data

members described below
  - Basic necessities for physics
    - double m_xVelocity
    - double m_yVelocity
    - double m_mass
  - Collision detection
    - CCollisionObject* standingOn - the CollisionObject that this MobileObject is currently standing on (if it's standing on anything).
  - fsf:CMutex m_csMobileObject - requisite Mutex
  - Member functions
    - bool collision(CCollisionObject* pColObj, bool &bTop, bool &bBottom, bool &bLeft, bool &bRight) const
      - Described in the detail in the physics module section below.

Modules

Modules:

- Graphics
- Music
- Networking
- Physics
- Control
- Gameplay

Graphics Modules

Render game team: Kathryn and Thea

Builder render team: My and Lilian

Platformer render team: Philip and Oliver

There are 3 graphics-related code modules.

The RenderGame? module (files `PtdRenderGameModule.[h|cc]`) defines the cell `ptd::CRenderGame` whose specifications are as follows:

`ptd::CRenderGame (fsf::CCell)`	PTD_RENDER_GAME
Active filter	[FSF_ACTIVE_PULSE "Root"]
Passive filter	[FSF_PASSIVE_FILTER "Root" [PTD_GAME_STATE "Game state" *GAME_STATE*] [OGL_CAMERA "Camera" *CAMERA*]]
Output	(no output)
	Produces a graphical representation of the current state of the game.

The basic RenderGame? module is used for debug rendering; it displays the entire map, all of the platformers, and all of the objects with basic graphics, also with the ability to zoom in and out and scroll around.

The BuilderRender? module (files `PtdBuilderRenderModule.[h|cc]`) defines the cell `ptd::CRenderBuilder` whose specifications are as follows:

`ptd::CRenderBuilder (fsf::CCell)`	PTD_RENDER_BUILDER
Active filter	[FSF_ACTIVE_PULSE "Root"]
	[FSF_PASSIVE_FILTER "Root" [PTD_GAME_STATE "Game state"

Passive filter	*GAME_STATE*] [OGL_CAMERA "Camera" *CAMERA*] [PTD_BUILDER_RENDER_PARAMETERS "Parameters" *PARAMETERS*]]
Output	(no output)
	Produces a graphical representation of the current state of the game for builder player interaction.

The PlatformerRender? module (files `PtdPlatformerRenderModule.[h|cc]`) defines the cell `ptd::CRenderPlatformer` whose specifications are as follows:

`ptd::CRenderPlatformer` (`fsf::CCell`)	PTD_RENDER_PLATFORMER
Active filter	[FSF_ACTIVE_PULSE "Root"]
Passive filter	[FSF_PASSIVE_FILTER "Root" [PTD_GAME_STATE "Game state" *GAME_STATE*] [OGL_CAMERA "Camera" *CAMERA*]]
Output	(no output)
	Produces a graphical representation of the current state of the game for platformer player interaction.

Textures are loaded only once in the prototype(x).cc or demo.cc file and their corresponding unsigned ints are saved in the CRenderPlatformer object. CPlatformer and CCollisionObject objects are then drawn and have corresponding textures applied to them.

Camera control is restricted to a box around the current client's CPlatformer object and is updated every pulse in the process function.

Music Module

Music Team: Chris Beavers, Sean Laguna, Steve Matsumoto

First, we get information from the game state. This is done by the **cell CAnalyzeGameState**, which gets information from the game state and sets flags that indicate whether or not a platformer has jumped, climbed, or completed the level, as well as the ID of the platformer.

The specifications for the `ptd::CAnalyzeGameState` cell are as follows:

`ptd::CAnalyzeGameState` (`fsf::CCell`)	PTD_ANALYZE_GAME_STATE
Active filter	[FSF_ACTIVE_PULSE "Root"]
Passive filter	[PTD_GAME_STATE "Game state"]
Output	(PTD_MIX_COMMAND "Command")
	Produces music/sound mix commands based on game state.

External methods called:

```
getJump()
getClimb()
getMoveUp()
getX()
getY()
```

The characteristics are stored in a **node** called a **CMixCommand** (which is created within the process method of CAnalyzeGameState). The data members of this CMixCommand include:

```
bool m_jump[5];
bool m_climb;
bool m_won;
```

```
std::string JUMP[6];
std::string CLIMB;
std::string WON;
```

The bools are set to true and false, depending on whether or not their corresponding audio track should be set to "playing." m_jump is an array of 5 flags, because each platformer has their own jump CAudioSample (we'll get into that later).

The strings are used by the getters and setters for these flags, to indicate which bool should be set. The 6th string in the JUMP string array is the generic string "jump" that doesn't correspond to any specific platformer, but indicates simply that the jump array is the data member we're dealing with. Then, the platformer ID determines which element in it to change.

The getter and setter thus have the following headings. whichS is which string ("jump" "climb" or "won") and whichP is the platformer ID (0-4).

```
/// getEffectFlag: returns the value of a specified effect flag
bool getEffectFlag(const std::string &whichS, const int &whichP);

/// setEffectFlag: sets the value of a specified effect flag
void setEffectFlag(const std::string &whichS, const int &whichP, const bool &val);
```

The setter is used by CAnalyzeGameState. The getter is used by **CApplyMixCommand**, a **cell** downstream from the CMixCommand node. This cell actually triggers the sound effects to be played. It iterates through all the CAudioSamples, checks whether or not their strings match any of the strings in CMixCommand, and then checks whether or not getEffectFlag thinks that sample should be playing. If it should, it sets the sample to play.

The specifications for the `ptd::CApplyMixCommand` cell are as follows:

`ptd::CApplyMixCommand (fsf::CCell)`	PTD_APPLY_MIX_COMMAND
Active filter	[PTD_MIX_COMMAND "Command"]
Passive filter	[PTD_AUDIO_LIBRARY "Audio library"]
Output	(no output)
	Applies music/sound mix commands to samples in the audio library.

The next **node** is **CAudioLibrary**. This stores a list of **CAudioSamples**, which store the actual audio data that can be played by C++. The CAudioSample class is totally separate from MFSM, and the CAudioLibrary acts as an interface between our CAudioSample class and the rest of the game.

CAudioLibrary has the following data members:

```
unsigned int m_nNbChannels; ///< Number of channels
std::list<CAudioSample*> m_vectSamples; ///< List of audio samples
```

All samples must have the same number of channels. m_vectSamples is an incredibly misleading name, because the samples are actually stored in a list.

CAudioLibrary has the following member functions:

```
/// mixes all the sound effects together by summing them and sending them to the
specified buffer
void mixSamples(audio::CAudioBuffer *pBuffer);

/// Adds a sample to the library
void addSample(CAudioSample *newSample);
```

```
/// Resets all sample play indices to zero
void resetSamples();

/// Returns a reference to the m_vectSamples
std::list<CAudioSample*>& getSamples();
```

CAudioSample has a constructor that takes in the name of a PCM wav file as a string and loads its data into the CAudioSample object. This is what we exclusively use to create audio samples. These objects have a bunch of data members that specify different things about the sample, but none of them are crucial for an outside understanding.

The important data members are these (there are many more):

```
double *m_pSampleDataArray; ///< Dynamically-allocated array of sample data
std::string m_strSampleName; ///< Name of sample
long m_nPlayIndex; ///< Play index
bool m_bIsPlaying; ///< A flag indicating whether the sample is playing
bool m_bIsLooping; ///< A flag indicating whether the sample is looping
```

The sample name corresponds to the strings addressed above, for comparison. m_nPlayIndex stores the spot in *m_pSampleDataArray that is being played currently. m_bIsPlaying is set to true if the effects trigger them in CApplyMixCommand's process function. m_bIsLooping is initially set to true for background music tracks, but not for sound effects.

Here is an abridged list of CAudioSample methods. The full list can be found in the header file.

```
/// Copies content at play index into argument buffer and advances play index
void playSample(audio::CAudioBuffer *pOutputBuffer);

/// Sets the flag indicating whether the sample is currently playing
void setIsPlaying(bool newIsPlaying);

/// Sets the flag indicating whether the sample is currently looping
void setIsLooping(bool newIsLooping);

/// Sets the play index
void setPlayIndex(unsigned int index);

/// Gets this audio sample's name
std::string getSampleName() const;

/// Gets the array of doubles representing the audio in the sample
double* data() const;
```

The final **cell** downstream from the CAudioLibrary node is **CMixAudio**. The only function in this class is its process function, which calls the CAudioLibrary class's function mixSamples()…and that's about all it does! It also sends the result of mixSamples as an output buffer, on the active pulse.

The specifications for the `ptd::CMixAudio` cell are as follows:

`ptd::CMixAudio (fsf::CCell)`	PTD_MIX_AUDIO
Active filter	[FSF_ACTIVE_PULSE "Root"]
Passive filter	[PTD_AUDIO_LIBRARY "Audio library"]
Output	(AUDIO_BUFFER "Mixed buffer")
	Produces mixed audio buffer.

And that's the end of our section of the code!

CELLS and NODES, in order:

CELL: CAnalyzeGameState (gets passive filter from game state)
NODE: CMixCommand
CELL: CApplyMixCommand (connected to CAudioLibrary repository, and to upstream cell CAnalyzeGameState)
NODE: CAudioLibrary (with a list of CAudioSamples)
CELL: CMixAudio (connected to CAudioLibrary repository)

Note: there are also AudioInputInterface and AudioOutputInterface nodes, and ReceiveAudio and SendAudio cells, respectively. They sandwich the CMixAudio cell, which sends the sound buffer as an active filter to the SendAudio cell.

Sound buffers are modeled as CAudioBuffer objects in MFSM.

Networking Module

Network team: Russell, Rahul, Richard

The system adopts a client server architecture in which the server receives commands from the player clients, performs game state updates, and sends updated game state information back to the clients.

Game server and clients communicate using the UDP protocol. Data flows from server to clients via multicast, and from clients to server via broadcast.

The Network module (files `PtdNetworkModule.[h|cc]`) defines elements that address basic communications and data encoding/decoding protocols.

Basic communications

* Class CConnection encapsulates CUDPSocket, established a communication line, works for receive and send, broadcast or multicast.

* Nodes: CUDPSocketInterface encapsulates one connection, supports receive and send, broadcast and multicast. CServerReceiveInterface manages receive connections with the 6 potential clients on the server side.

* Cells

`ptd::CUDPSend (fsf::CCell)`	PTD_UDP_SEND
Active filter	[FSF_CHAR_BUFFER "Message"]
Passive filter	[FSF_ACTIVE_PULSE "Root" [PTD_UDP_SOCKET_INTERFACE "Network output interface" *SOCKET_INTERFACE*] [PTD_NETWORK_CONNECTION "Network connection" *NETWORK_CONNECTION*]]
Output	(no output)
	Sends the data in the buffer received in the active pulse to the address and port specified in the communication node through the socket interface.

.

`ptd::CUDPSendFromQueue (fsf::CCell)`	PTD_UDP_SEND_FROM_QUEUE
Active filter	[FSF_CHAR_BUFFER "Message"]
Passive filter	[FSF_ACTIVE_PULSE "Root" [PTD_UDP_SOCKET_INTERFACE "Network output interface" *SOCKET_INTERFACE*] [PTD_NETWORK_CONNECTION "Network connection" *NETWORK_CONNECTION*]

	[PTD_MESSAGE_QUEUE "Message queue" *MESSAGE_QUEUE*]]
Output	(no output)
	Removes the next message in the queue and sends to the address and port specified in the communication node through the socket interface.

.

`ptd::CUDPReceive (fsf::CCell)`	PTD_UDP_RECEIVE
Active filter	(not used)
Passive filter	[PTD_UDP_SOCKET_INTERFACE "Network input interface"]
Output	[FSF_CHAR_BUFFER "Message"]
	Places data received from the socket interface on the active pulse.

Command vocabulary and coding

Keyboard commands performed by the clients (i.e. key down and up events) are intercepted by the CDispatchKeyboardEvent, and then specific useful keys (e.g. w, a, s, d) are converted into CharBuffers?, which are then sent to the server. Mouse commands can be intercepted and transfered by CDispatchMouseEvent, but doing it this way led to issues (i.e. mouse events from the builder are often meaningless, because the server does not know where on the map it is clicking, due to a moving camera that it is not aware of). As such, the code for this exists but is not used in the current implementation.

When the server receives these events, it uses CDecodeEvent to transform the CharBuffers? that it receives into Keyboard/Mouse events for specific clients, and then passes these events to the clients respective control modules. All control processing that affects the Game State is performed on the server.

This protocol is encoded in a number of cells and one

`ptd::CDispatchKeyboardEvent (fsf::CCell)`	PTD_DISPATCH_KEYBOARD_EVENT
Active filter	[GLUTIO_KEYBOARD_EVENT "Event"]
Passive filter	[PTD_MESSAGE_QUEUE "Message queue"]
Output	(GLUTIO_KEYBOARD_EVENT "Local event")
	Dispatches keyboard events received on the active stream: local events (e.g. rendering control) are echoed with a different name, play control events are encoded and the resulting buffer is placed in a message queue for transmission.

.

`ptd::CDispatchMouseEvent (fsf::CCell)`	PTD_DISPATCH_MOUSE_EVENT
Active filter	[GLUTIO_MOUSE_EVENT "Event"]
Passive filter	[PTD_MESSAGE_QUEUE "Message queue"]
Output	(GLUTIO_MOUSE_EVENT "Local event")
	Dispatches mouse events received on the active stream: local events (e.g. rendering control) are echoed with a different name, play control events are encoded and the resulting buffer is placed in a message queue for transmission.

.

ptd::CDecodeEvent (fsf::CCell)	PTD_DECODE_EVENT
Active filter	[FSF_CHAR_BUFFER "Message"]
Passive filter	[FSF_PASSIVE_PULSE "Root"]
Output	(GLUTIO_KEYBOARD_EVENT "Keyboard event") or (GLUTIO_MOUSE_EVENT "Mouse event")
	Decodes message received on the stream into a keyboard or mouse event. The output name gives indication of originating client (type of player and ID).

Game state coding

The Game State team provided two very nice functions, one that converts the game state into a CharBuffer?, and one that converts a "proper" CharBuffer? into a game state. Thus, we have the following two tasks.

CEncodeGameState takes the game state on the server, turns it into the CharBuffer?, adds some other important information to this CharBuffer?, and then sends this to the multicast group, where all of the clients get it from.

CUpdateGameState takes some CharBuffer? that it receives from the multicast group (note: this only is used in the Client), and transforms the current game state into the new game state by applying this CharBuffer?.

ptd::CEncodeGameState (fsf::CCell)	PTD_ENCODE_GAME_STATE
Active filter	[FSF_ACTIVE_PULSE "Root"]
Passive filter	[PTD_GAME_STATE "Game state"]
Output	(FSF_CHAR_BUFFER "Message")
	Encodes the game state into a character buffer according to the protocol described above, and places on the stream for network transmission.

.

ptd::CUpdateGameState (fsf::CCell)	PTD_UPDATE_GAME_STATE
Active filter	[FSF_CHAR_BUFFER "Message"]
Passive filter	[PTD_GAME_STATE "Game state"]
Output	(no output)
	Updates the game state in the repository with the game state encoded in the character buffer received on the stream.

.

Physics Module

Physics team: Max, Bea and Leif

The Physics module (files `PtdPhysicsModule.[h|cc]`) defines the cell `ptd::CStepPhysics` whose specifications are as follows:

ptd::CStepPhysics	PTD_STEP_PHYSICS

(fsf::CCell)	
Active filter	[FSF_ACTIVE_PULSE "Root"]
Passive filter	[PTD_GAME_STATE "Game state"]
Output	(no output)
	Applies one step in the dynamic physics simulation for all objects subject to physics in the game state.

Implementation

The module also defines a hierarchy of classes for modeling (virtual) world objects:

`CCollisonObject` - a basic object in the game world, this is the class all other objects are derived from. A `CCollisionObject` has a position, dimensions, color, identification number, and a coefficient of friction. A collision object is immobile, but it can be collided with by a mobile object

`CMobileObject` - an object derived from a CCollisionObject that is affected by physics. A `CMobileObject` will fall if it is not on a stationary surface and if it is pushed it will move until stopped by friction. In addition to the information inherited from `CCollisionObject`, a `CMobileObject` has a velocity, mass, and a pointer to the object it is sitting on, if applicable. The `CMobileObject` class also contains a function to determine if the object has collided with a given object.

`CPlatformer` - an object derived from a `CMobileObject` that represents a platformer player in the game. A `CPlatformer` can be moved by the user and has a set of rules to limit its movement. In addition to the information inherited from `CMobileObject`, a `CPlatformer` has a maximum velocity and acceleration, flags indicating the movement the user is requesting, and a variable that allows the physics module to process variable jump height

The majority of the work for the physics is done in the process() function. This function iterates through lists of all the `CMobileObjects` and `CPlatformers` in the game world and updates their position, velocity, and other properties. Once the position of an object is adjusted, it is tested for collision with other objects, and if there is a collision the object is moved to the bounds of the object it hit and its velocity is adjusted. The process() function also interprets information about the action of the user and adjusts the `CPlatformer`'s velocity and acceleration accordingly.

Another large portion of the physics work is done in the collision function, located in `CMobileObject`. The signature for this is:

```
bool collision(CCollisionObject *pColObj, bool &bTop, bool &bBottom, bool &bLeft,
bool &bRight) const;
```

The collision function first checks if the calling `CMobileObject` overlaps at all with the given `CCollisionObject`, and immediately halts and returns false if there is no overlap. If there is overlap, the collision function gets the velocity of the `CMobileObject` and finds the corner of the object that points in the direction of movement, as well as the corner of the of the `CCollisionObject` that points toward the mobile object. A line is then drawn between the corner of the mobile object in its current position and the same corner of the mobile object in its previous position. This line is then analyzed to see if it passes above or below the chosen corner of the `CCollisionObject`. The result of this analysis determines the side of the `CMobileObject` on which the collision occurred. One of the given flags is then updated to reflect the side the collision occurred on.

Control Module

There are 3 control-related code modules: a basic control module for debugging and spectator clients, a platformer control module, and a builder control module.

Basic control module

The basic Control module (files `PtdControlModule.[h|cc]`) defines the cells `ptd::CKeyboardControl` and `ptd::CMouseControl` whose specifications are as follows:

`ptd::CKeyboardControl` (`fsf::CCell`)	PTD_KEYBOARD_CONTROL
Active filter	[GLUTIO_KEYBOARD_EVENT "Keyboard event"]
Passive filter	[FSF_PASSIVE_FILTER "Root" [PTD_GAME_STATE "Game state" *GAME_STATE*] [OGL_CAMERA "Camera" *CAMERA*]]
Output	(no output)
	Processes keyboard event and applies corresponding action to game state and/or camera for game state rendering. This cell implements keyboard interaction for debugging the game state and for spectator clients in conjunction with the RenderGame module.
`ptd::CMouseControl` (`fsf::CCell`)	PTD_MOUSE_CONTROL
Active filter	[GLUTIO_MOUSE_EVENT "Mouse event"]
Passive filter	[FSF_PASSIVE_FILTER "Root" [PTD_GAME_STATE "Game state" *GAME_STATE*] [OGL_CAMERA "Camera" *CAMERA*]]
Output	(no output)
	Processes mouse event and applies corresponding action to game state and/or camera for game state rendering. This cell implements mouse interaction for debugging the game state and for spectator clients in conjunction with the RenderGame module.

Keyboard mappings:

= Key =	= Effect =
F	Switch to full screen mode
f	Switch to window mode
=	Zoom in
-	Zoom out
w	Move view up
s	Move view down
a	Move view left
d	Move view right
g	View whole screen
<ESC>	quit

Mouse mappings:

- Click and drag to move map view.

Builder control module

(Builder Controls: David, Sarah, and Heather)

The Builder control module (files `PtdBuilderControlModule.[h|cc]`) defines the cells `ptd::CKeyboardControlBuilder` and `ptd::CMouseControlBuilder` whose specifications are as follows:

`ptd::CKeyboardControlBuilder` (`fsf::CCell`)	PTD_KEYBOARD_CONTROL_BUILDER

Active filter	[GLUTIO_KEYBOARD_EVENT "Keyboard event"]
Passive filter	[FSF_PASSIVE_FILTER "Root" [PTD_GAME_STATE "Game state" *GAME_STATE*] [OGL_CAMERA "Camera" *CAMERA*]]
Output	(no output)
	Processes keyboard event and applies corresponding action to builder player and/or camera for builder-centric rendering. This cell implements keyboard interaction for builder player in conjunction with the RenderBuilder module.
`ptd::CMouseControlBuilder` (`fsf::CCell`)	PTD_MOUSE_CONTROL_BUILDER
Active filter	[GLUTIO_MOUSE_EVENT "Mouse event"]
Passive filter	[FSF_PASSIVE_FILTER "Root" [PTD_GAME_STATE "Game state" *GAME_STATE*] [OGL_CAMERA "Camera" *CAMERA*]]
Output	(no output)
	Processes mouse event and applies corresponding action to builder player and/or camera for builder-centric rendering. This cell implements mouse interaction for builder player in conjunction with the RenderBuilder module.

Keyboard mappings:

= Key =	= Effect =
F	Switch to full screen mode
f	Switch to window mode
=	Zoom in
-	Zoom out
w	Move view up
s	Move view down
a	Move view left
d	Move view right
g	View whole screen
1	Select Item 1 (a normal red block)
2	Select Item 2 (a blue block with a lot of friction)
3	Select Item 3 (a green wall that is unclimbable)
<ESC>	quit

Mouse mappings:

- In main map screen:
  - Click and drag to move map view.
  - If an item is selected, and the builder has enough money, then click to place the item.
- In mini map screen:
  - Click on a part of the mini map to move main map view to that area.
- In item toolbar screen:
  - Click on arrow buttons to cycle through possible items.
  - Click on item button to select that item.

Platformer control module

(Platformer Controls: Aaron and Kevin)

The Platformer control module (files `PtdBuilderControlModule.[h|cc]`) defines the cells `ptd::CKeyboardControlPlatformer` and `ptd::CMouseControlPlatformer` whose specifications

are as follows:

`ptd::CKeyboardControlPlatformer (fsf::CCell)`	PTD_KEYBOARD_CONTROL_PLATFORMER
Active filter	[GLUTIO_KEYBOARD_EVENT "Keyboard event"]
Passive filter	[FSF_PASSIVE_FILTER "Root" [PTD_GAME_STATE "Game state" *GAME_STATE*] [OGL_CAMERA "Camera" *CAMERA*]]
Output	(no output)
	Processes keyboard event and applies corresponding action to platformer player and/or camera for platformer-centric rendering. This cell implements keyboard interaction for platformer player in conjunction with the RenderPlatformer? module.
`ptd::CMouseControlPlatformer (fsf::CCell)`	PTD_MOUSE_CONTROL_PLATFORMER
Active filter	[GLUTIO_MOUSE_EVENT "Mouse event"]
Passive filter	[FSF_PASSIVE_FILTER "Root" [PTD_GAME_STATE "Game state" *GAME_STATE*] [OGL_CAMERA "Camera" *CAMERA*]]
Output	(no output)
	Processes mouse event and applies corresponding action to platformer player and/or camera for platformer-centric rendering. This cell implements mouse interaction for platformer player in conjunction with the RenderPlatformer? module.

Keyboard mappings:

= Key =	= Effect =
S	Switch to full screen mode
s	Switch to window mode
<ESC>	quit

Mouse mappings: None for now...

Gameplay Module

Enforces the rules, i.e. your health and its effects, whether or not you've made it to the goal, etc.

Will probably have a different one for platformer and builder.

Appendix A - Getting Started with the Code

- Download `mfsm_1.0-r1b.zip` and uncompress in a directory named `cs121-project` hereafter (the name is not important).
    - rename `mfsm_1.0-r1b` to `MFSM_1.0`
    - in a Terminal window, go to directory `Fsf`, run `make`
    - go to each *xxx*`Example` directory and run `make`, or `run -f Makefile.macosx` as appropriate
    - to run the examples, from the appropriate example directory, run `./Darwin_i386/bin/`*xxx*

(e.g. from `UserGuideExamples` run `./Darwin_i386/bin/example1`)

- Move the `RtAudio` directory from `RtAudioExamples` up to the `cs121-project` directory: Platitude will use `RtAudio` as well...). Edit `RtAudioExamples/Makefile.macosx` to update the path to the `RtAudio` directory and check that the examples compile and run...

- Checkout the Platitude code from the class SVN repository into the `cs121-project` directory:
  - in a Terminal window, from the cs121-project directory, run: `svn checkout https://svn.cs.hmc.edu/svn/cs121fa2010 Platitude`
  - see Using Subversion (svn) as needed

The cs121-project directory now contains directories `MFSM_1.0`, `RtAudio`, and `Platitude`.

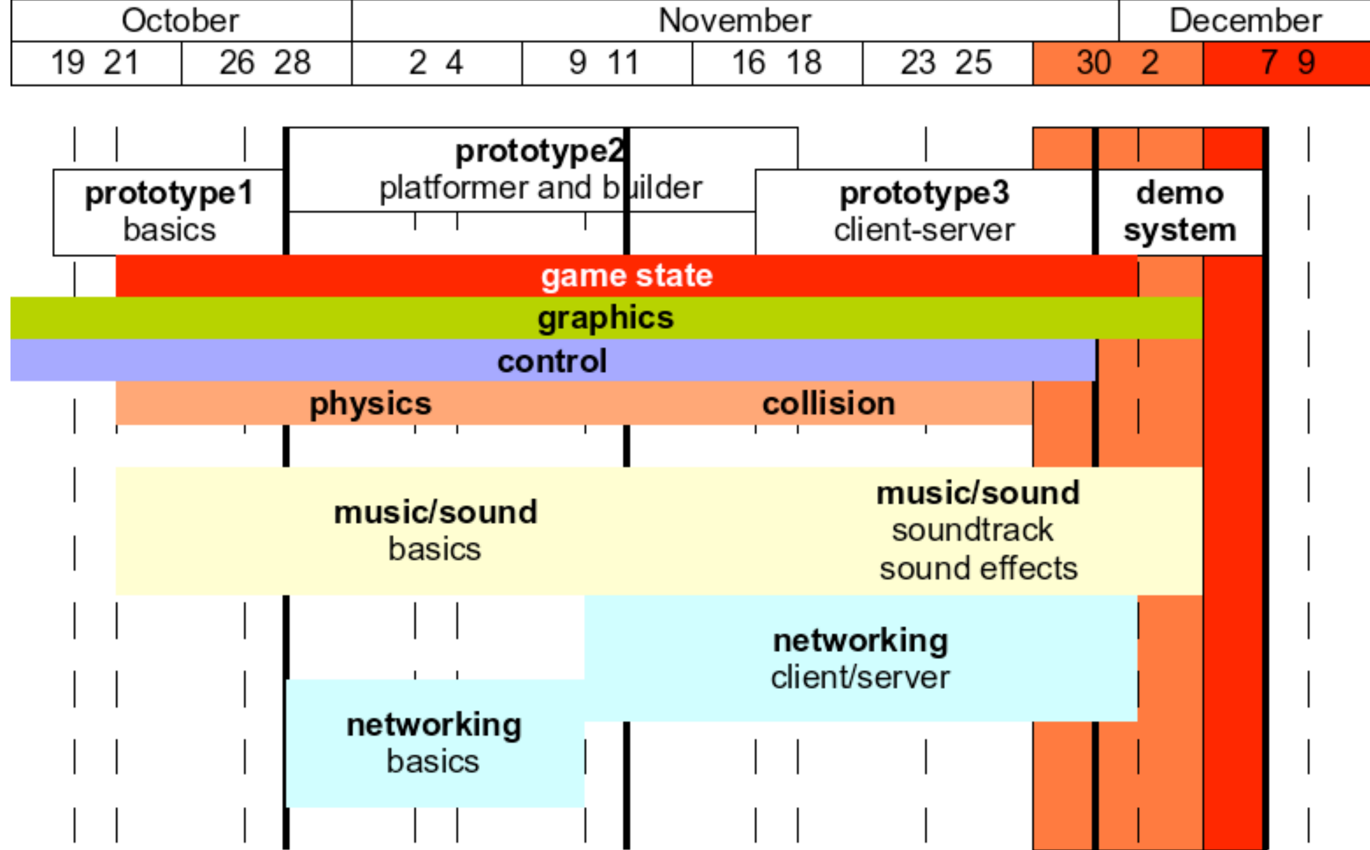
Platitude is the working directory for the project. You should not have to touch anything else.

- Go to the `Platitude` directory and run `make -f Makefile.macosx`
- Run `./Darwin_i386/bin/prototype1 -i 0 -o 2`

- Start messing with the code!
- Follow svn basic work cycle

Appendix B - Development Plan and Schedule

Schedule



Milestones

Prototype 1: the basics

Target date: 28 October 2010

Prototype 1 implements initial versions of game state elements, game state rendering (for debugging) and interactive control of the view, basic physics (dynamics).

Music/sound and networking aspects should be investigated separately.

Prototype 1 system graph: interaction

Physics
Graphics
Control

Pulsar
pulse rate
+ empty pulse

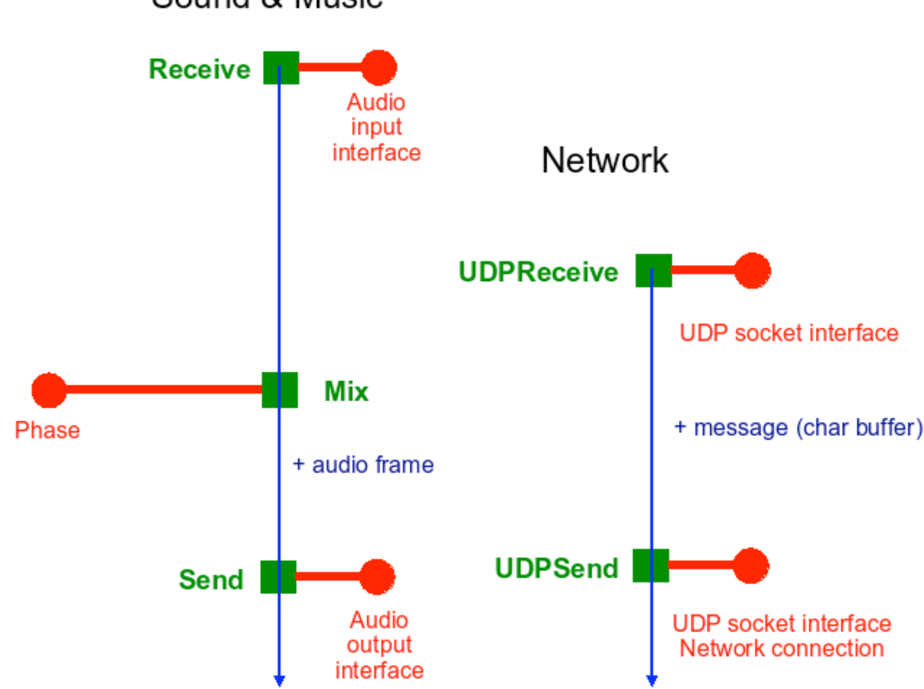
Pulsar
render rate
+ empty pulse

ListenKeyboard
+ keyboard event

ListenMouse
+ mouse event

StepPhysics
Game state
Rendering camera and parameters
RenderGame
Updates rendering parameters only
KeyboardControl
MouseControl

Prototype 1 system graph: sound & music, network I/O



Prototype 2: platformer and builder

Target date: 11 November 2010

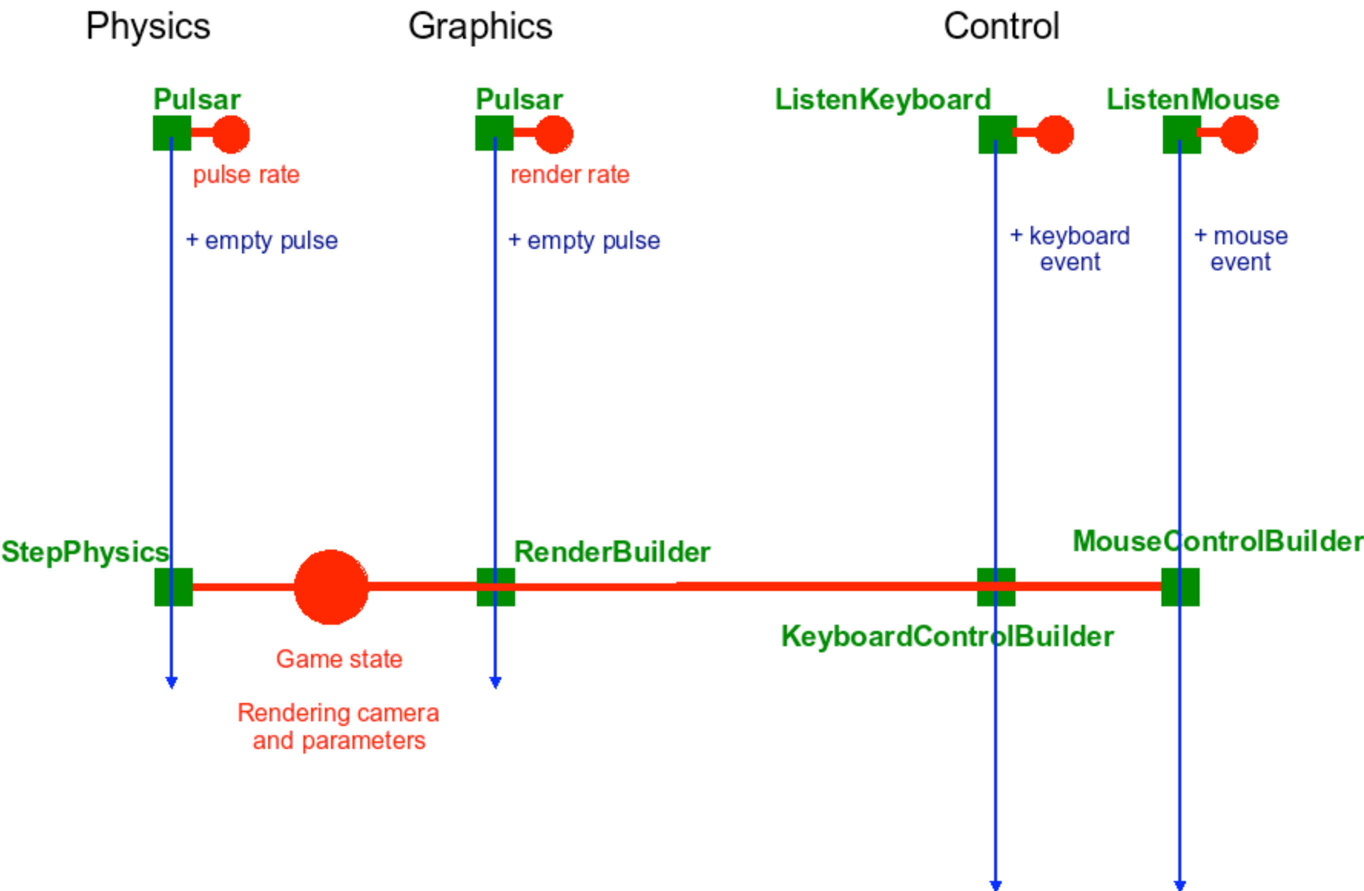
Prototype 2 implements two stand-alone systems: builder and platformer. Each integrates rendering and controls specific to the player type. The game state now captures most of the

elements needed for game mechanics. Both systems implement collision detection and basic sound effects.

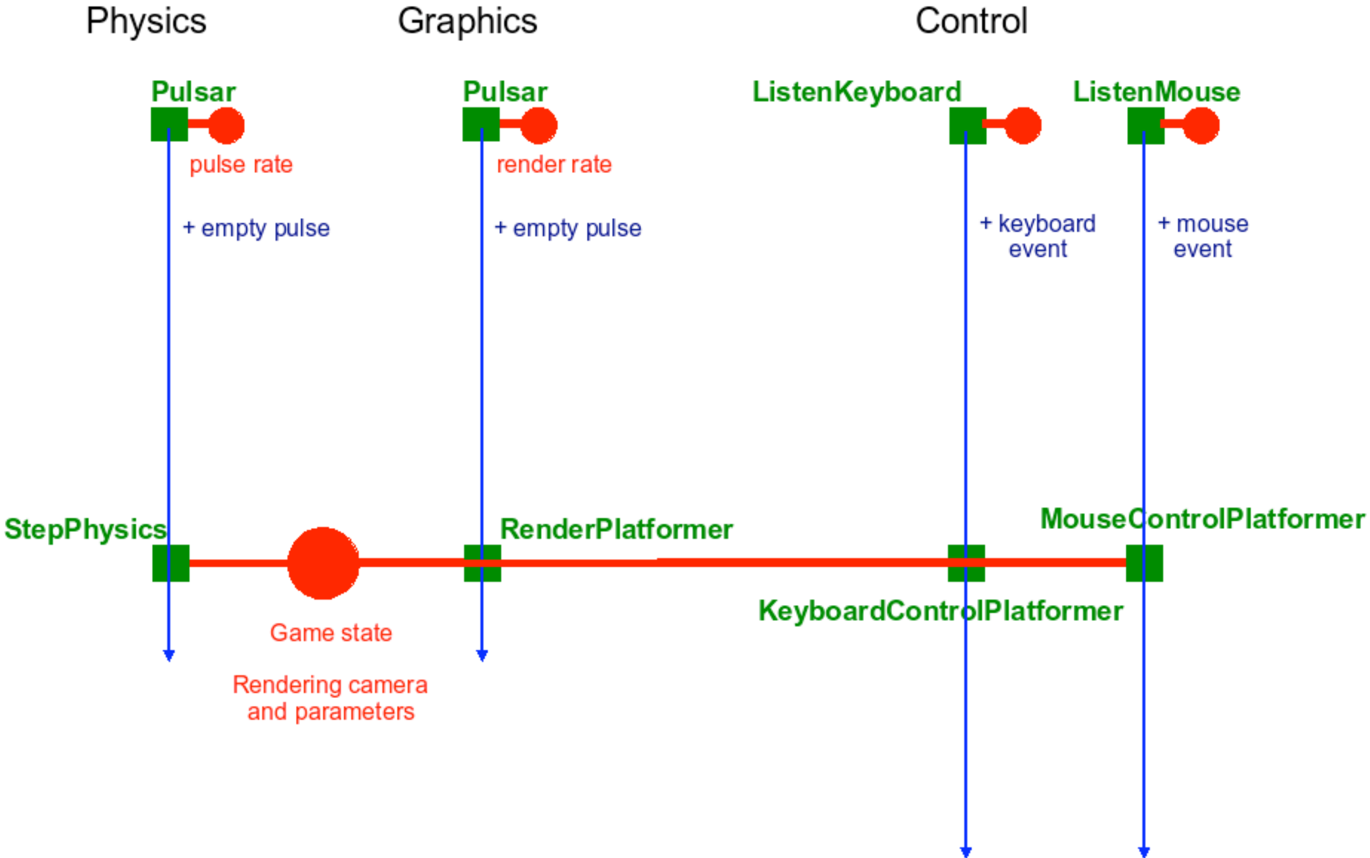
Networking development remains separate.

Prototype 1 remains as a testing and debugging platform (spectator role).

Prototype 2 system graph: builder



Prototype2 system graph: platformer



Prototype 3: client-server

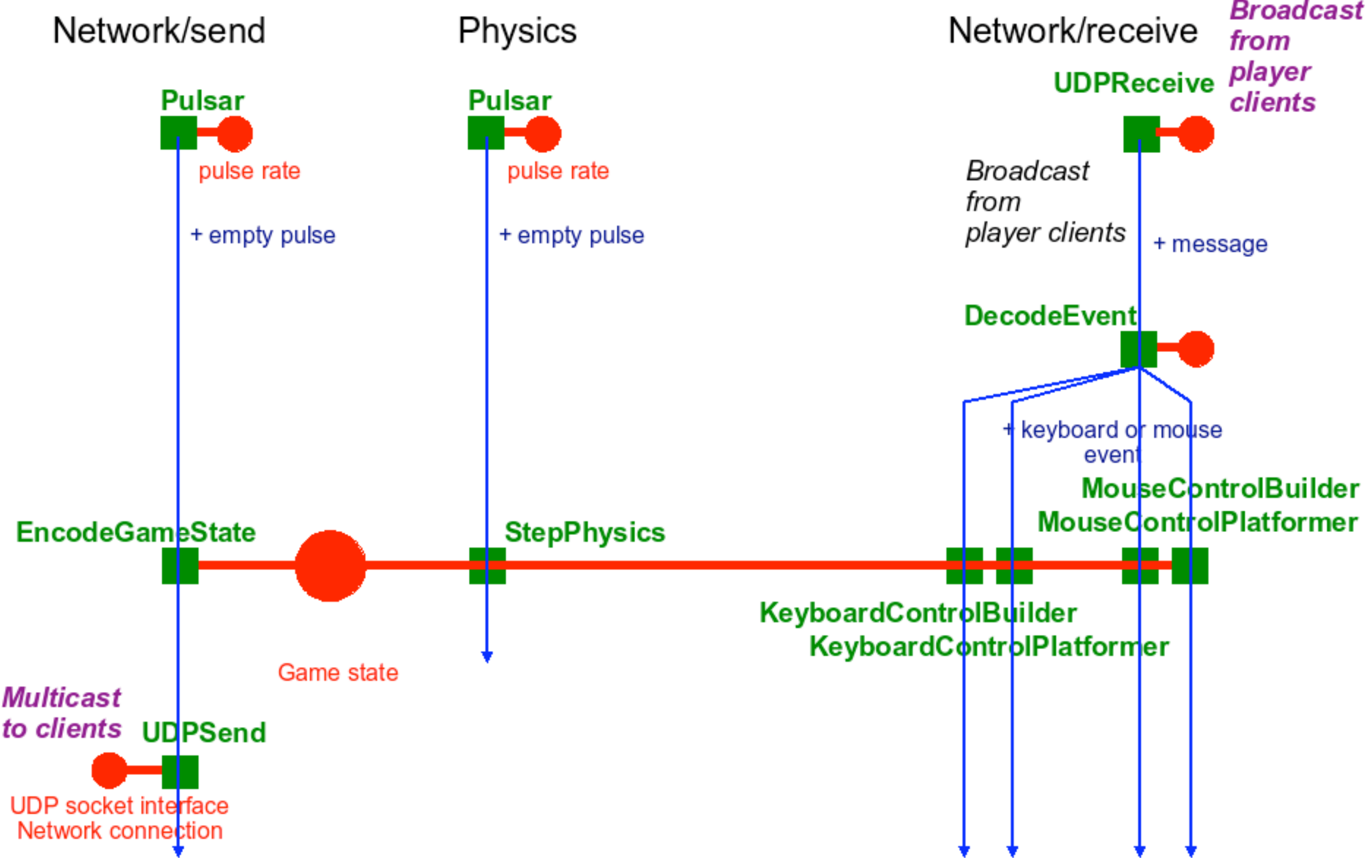
Target date: 25 November 2010

Note: Prototype 3's design cannot accommodate builder controls, which involve interaction between graphics and control modules. A solution would involve adding an intermediate level of indirection in which control action are encoded into commands containing all necessary information, which would then be sent over the network (rather than the raw keyboard or mouse events) to be decoded and applied on the server side. The demo system will compensate for this design flaw by combining game server and builder.

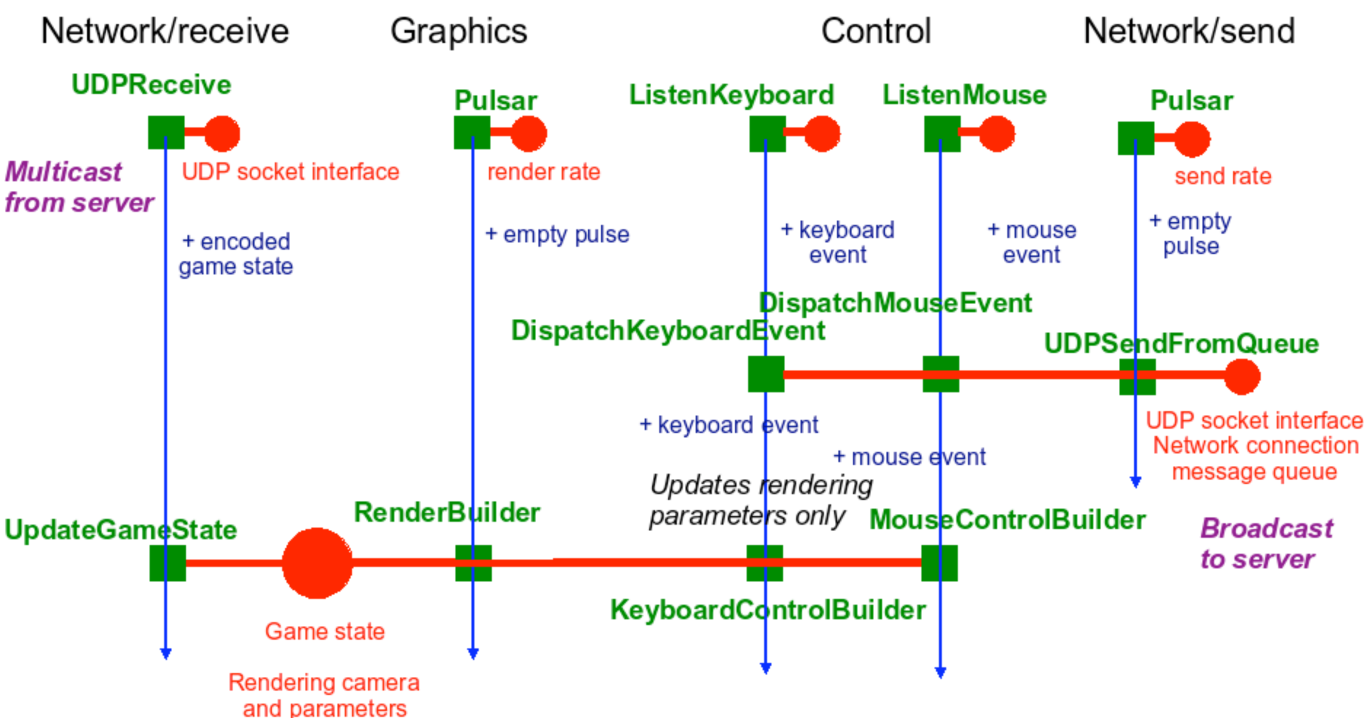
Prototype 3 implements the client-server architecture: a game server, and player and spectator clients. The game state encompasses all relevant information for game mechanics and logic. The systems implement basic game logic, simple AI elements, contextual sound effects and music.

Prorotype 1 and 2 remain as testing and debugging platformer (physics, network I/O, builder and platformer interaction)

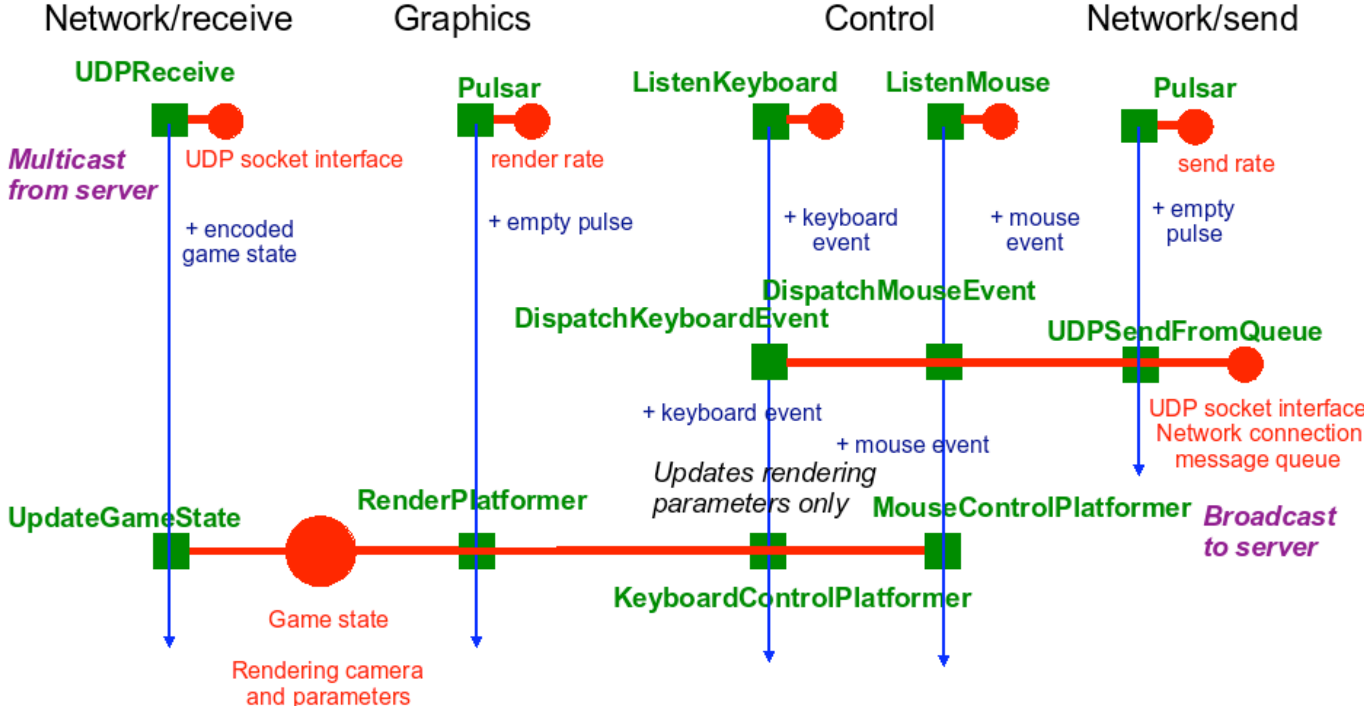
Prototype 3 system graph: game server



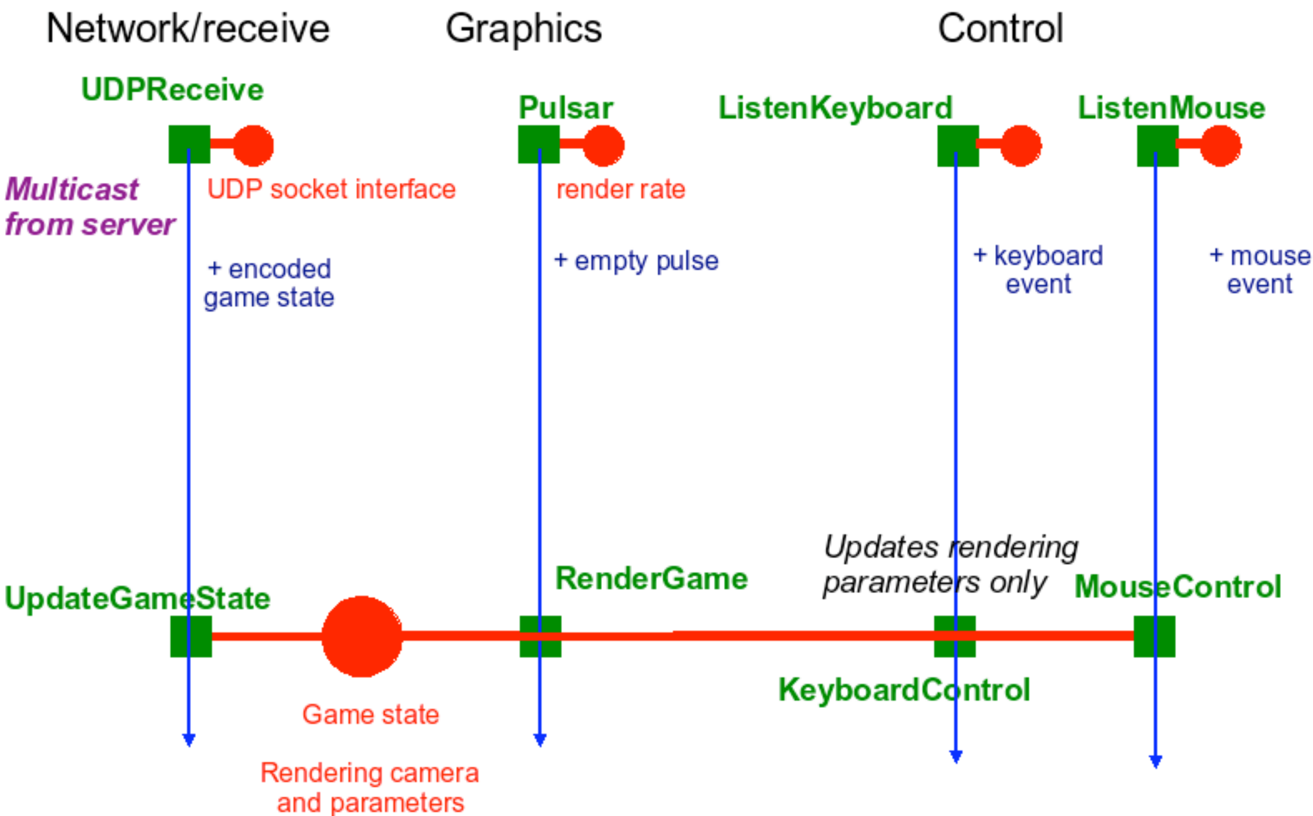
Prototype 3 system graph: builder client



Prototype 3 system graph: platformer client



Prototype 3 system graph: spectator client

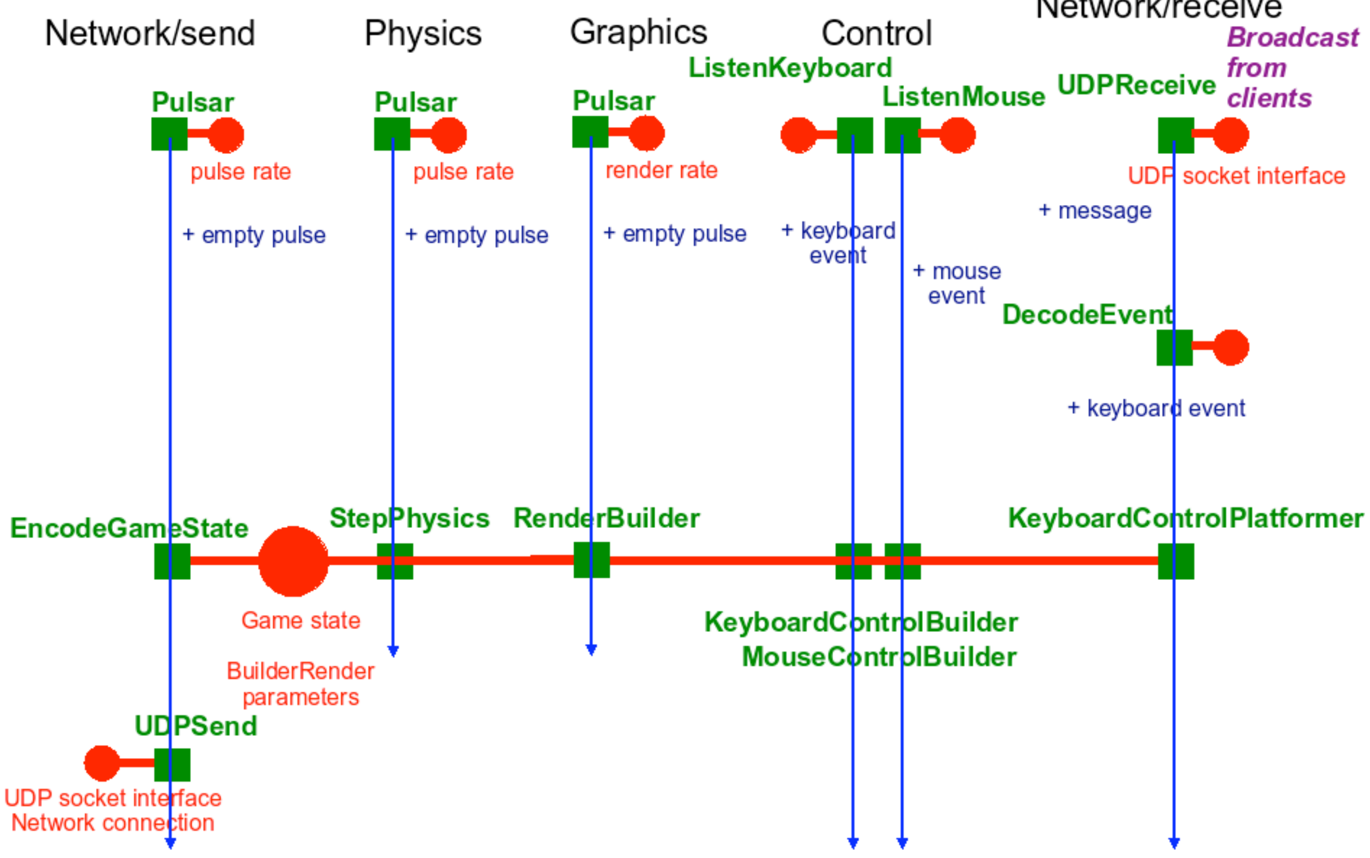


Demo system

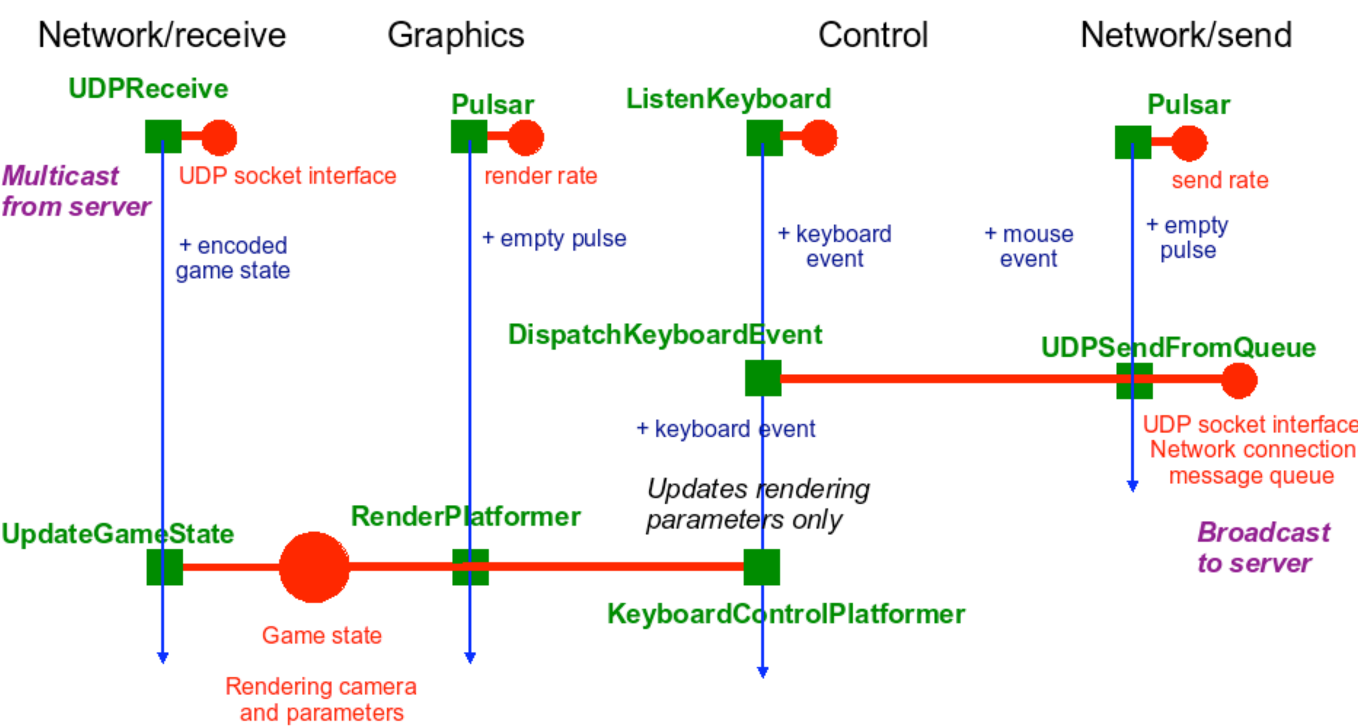
Target date: 7 December 2010

The demo system integrates some architectural adjustments to compensate for a design flaw in Prototype 3.

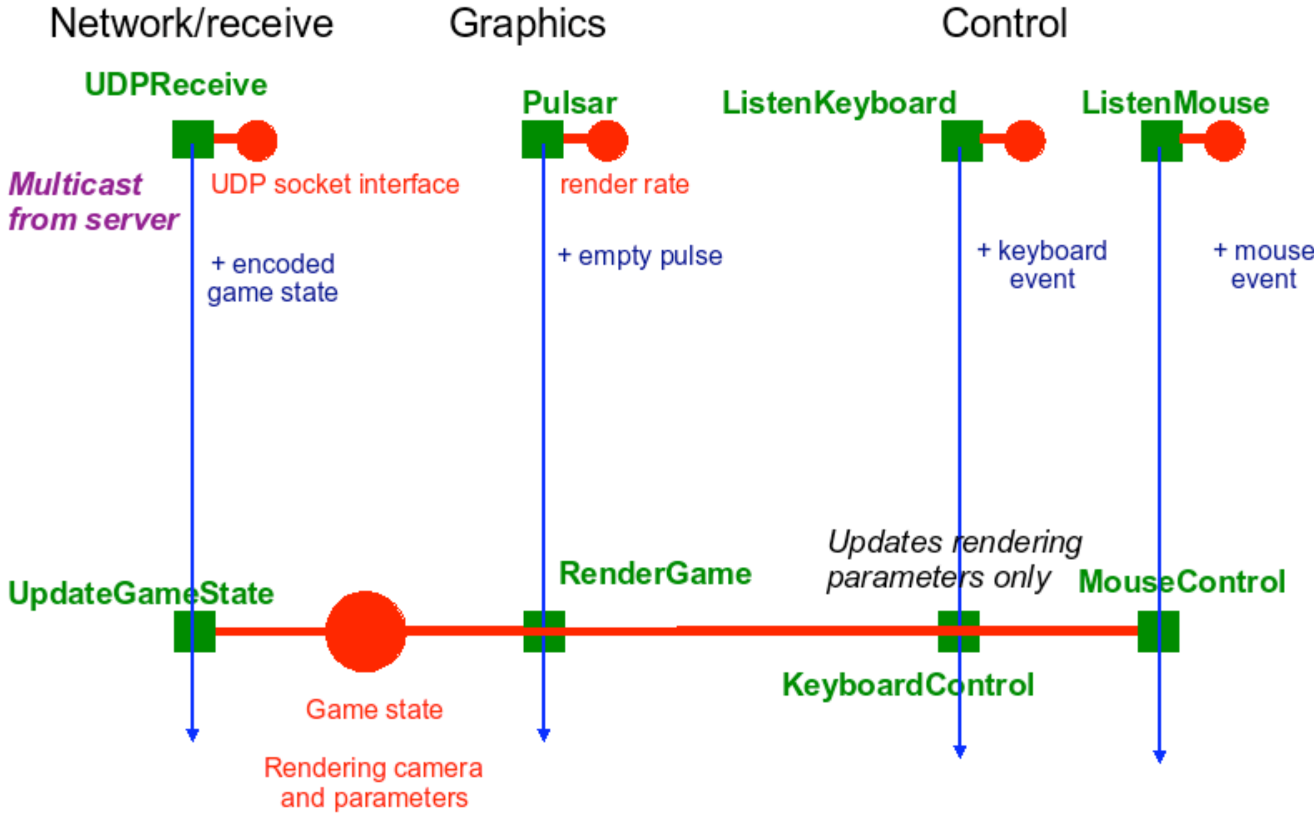
Demo system graph: game server + builder



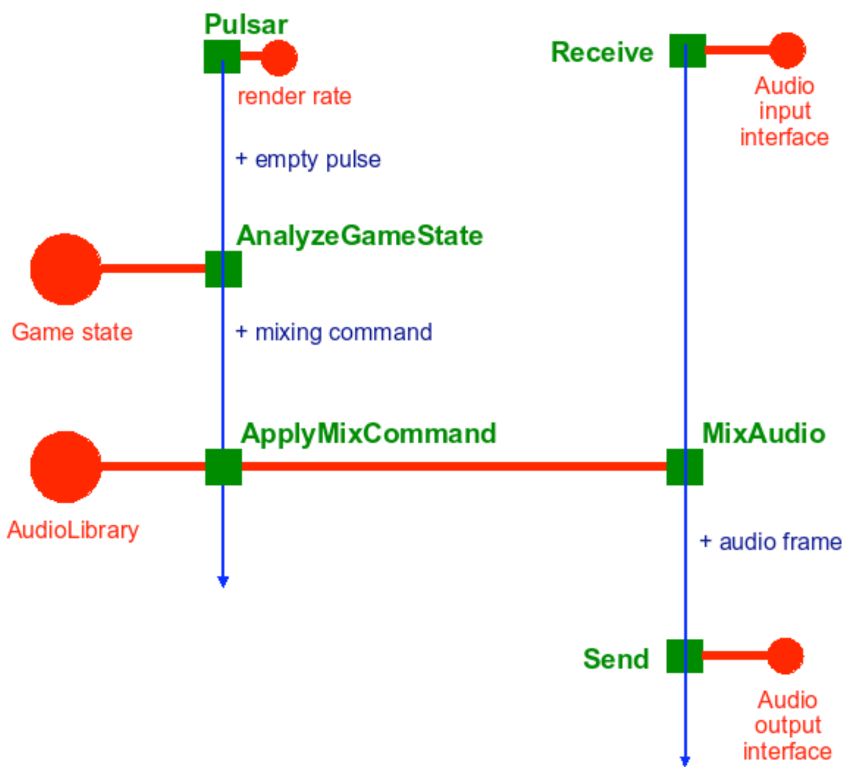
Demo system graph: platformer client



Demo system graph: spectator client



Demo system graph: sound and music subsystem



Attachments

- prototype1-interaction.png (52.7 KB) - added by *alex* 5 weeks ago. "Prototype 1 system graph - interaction"
- prototype1-sound-music.png (20.7 KB) - added by *alex* 5 weeks ago. "Prototype 1 system graph - sound & music"

- prototype1-network.png (21.0 KB) - added by *alex* 5 weeks ago. "Prototype 1 system graph - network"
- prototype2-builder.png (48.0 KB) - added by *alex* 5 weeks ago. "Prototype 2 system graph - builder"
- prototype2-platformer.png (49.7 KB) - added by *alex* 5 weeks ago. "Prototype 2 system graph - platformer"
- prototype3-spectator.png (62.9 KB) - added by *alex* 5 weeks ago. "Prototype 3 system graph - spectator client"
- prototype3-builder.png (98.4 KB) - added by *alex* 5 weeks ago. "Prototype 3 system graph - builder client"
- prototype3-platformer.png (99.7 KB) - added by *alex* 5 weeks ago. "Prototype 3 system graph - platformer client"
- prototype3-server.png (85.3 KB) - added by *alex* 5 weeks ago. "Prototype 3 system graph - game server"
- demo-gb.png (90.1 KB) - added by *alex* 3 weeks ago. "Demo system graph - game server and builder"
- demo-p.png (88.9 KB) - added by *alex* 3 weeks ago. "Demo system graph - platformer client"
- demo-s.png (62.9 KB) - added by *alex* 3 weeks ago. "Demo system graph - spectator client"
- demo-m.png (36.9 KB) - added by *alex* 3 weeks ago. "Demo system graph - sound and music subsystem"
- schedule.png (57.5 KB) - added by *alex* 12 days ago. "Project schedule"