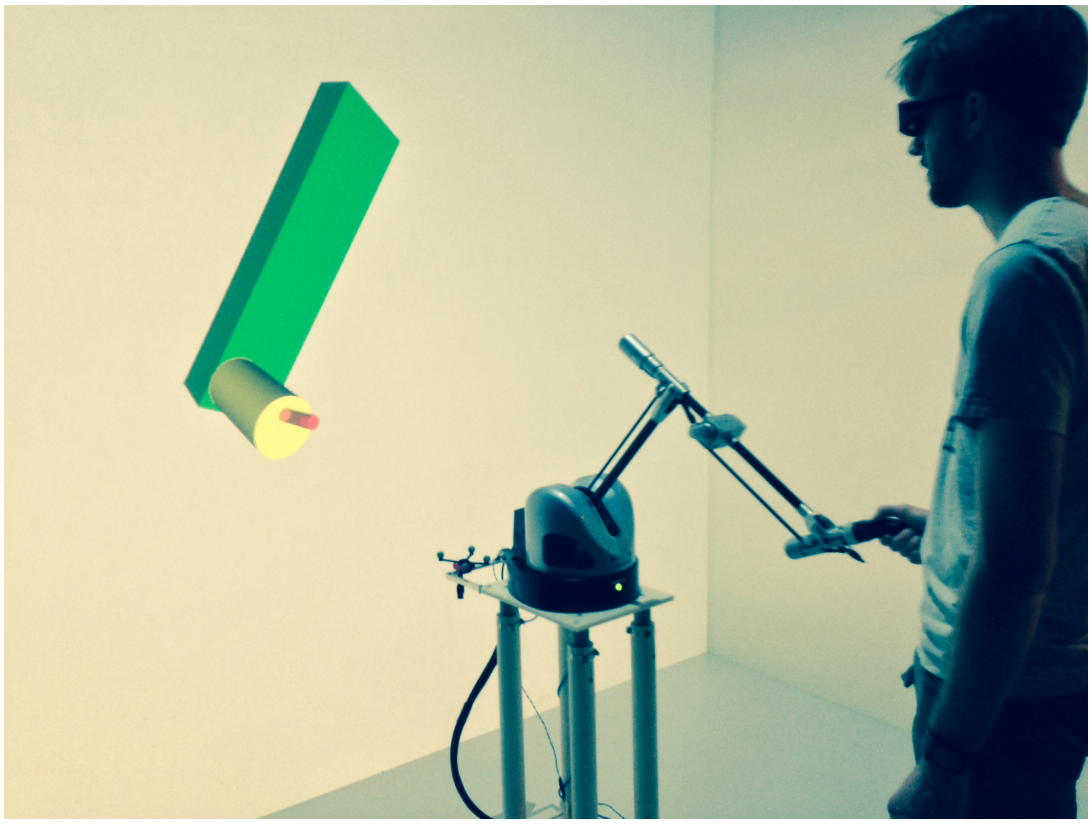
Collision Detection and Teamcenter Haptics: CATCH

Final Report

May 14-30

Logan Scott
Matt Mayer
James Erickson
Anthony Alleven
Paul Uhing



Acknowledgements

Thank you to all that helped us achieve our goal. This includes ,but is not limited to, Dr. Vance (VRAC, Iowa State University), Pin (Siemens), Dr. Weiss (Iowa State University), Jerome (Haption), and other stakeholders who gave us useful input throughout the project.

Table of Contents

1. Introduction

1.1. Purpose

This document contains relevant information including the design, implementation, and functionality of CATCH. CATCH is a standalone library for integrating 3D modeling software with a haptic device, developed as a senior design project in the Department of Electrical and Computer Engineering at Iowa State University. This document provides insight into the design process of the group as well as describes both high-level functionality and low-level design of our project for use by anyone who would want to build upon our work.

1.2. Problem Statement

The field of virtual reality has advanced greatly in the past few years. One factor continuing to slow the advancement of virtual reality is its lack of reality. Haptic technology is adding this missing reality to the virtual world by introducing tactile feedback to virtual interactions. This allows the user to experience virtual object through the use of force feedback, vibration, or motion. While researchers like Dr. Vance have had success making use of these devices, many companies that would like to use this technology do not try to use haptic technology in their design process for two reasons. The first is that the devices themselves are very expensive. The second issue is a lack of commercial software support for the integration of haptic technology with existing 3D modeling software. Our project is a proof of concept demonstrating integration of TeamCenter Visualization, a physics engine, a haptic arm, and 3D model file types. TeamCenter Visualization is a common commercial application used to model 3D objects. Our application will demonstrate this integration by allowing a user to interact with the haptic arm to manipulate a cursor in Teamcenter Visualization, manipulate parts in the scene using the cursor, and allowing collision detection to occur between parts. Based on this collision detection, the user will experience haptic feedback such that they are unable to push a part “through” another part.

1.3. Requirements

1.3.1. Functional

1.3.1.1. The cursor in the 3D visualizer will be manipulated by a haptic device.

1.3.1.2. Parts of the 3D assembly can be selected, deselected, and moved by the chosen haptic device.

1.3.1.3. The parts in the scene will be loaded into the said scene from a standard 3D modeling file type.

1.3.1.4. Collisions between parts in the scene must be detected and appropriate haptics feedback should be provided.

1.3.1.5. The library must run on a Windows 7 x64 environment.

1.3.1.6. CATCH must be capable of interacting with 3D models containing

simple geometry.

1.3.1.7. The library must run in the CAVE environment in the METaL lab at Iowa State.

1.3.2. Non-Functional

1.3.2.1. Teamcenter Visualization Mockup must be used as the Visualizer.

1.3.2.2. VisController API must be used to interface with the 3D Visualizer.

1.3.2.3. JT Open Toolkit must be used to support loading of part geometry via Jupiter Tessellation (JT) files.

1.3.2.4. Must use the IPSI physics engine from Haption for collision detection and interactions with the haptic device.

1.3.2.5. CATCH must interface with the Virtuose haptic device from Haption.

1.3.2.6. The lag time between input and output shall be less than 200ms for best user experience.

1.3.2.7. All public modules and functions shall be documented to the extent at which they could be recreated by a third party.

1.3.2.8. After accounting for lag time, all object models shall be synchronized.

1.3.3. Use Case

The CATCH library has two main use cases. It could be integrated in to future projects created by VRAC students. The second use case is for in-house testing of Siemens VisController API.

1.3.3.1. VRAC students who are working with the Virtuose haptic device or other similar devices could use our library to integrate new capabilities for use with the METaL Cave environment.

1.3.3.2. Siemens is currently in the process of developing the VisController API used in this project. The CATCH library could be used for in house testing of this API with the goal of eventually making a version of VisController API that would allow companies to make use of haptic technologies using Siemens software.

1.4. Dependencies

This section describes external libraries or applications that were not developed by the CATCH team. In other words, the functionality of the following libraries and functions are used by the CATCH modules, but the CATCH modules are in no way responsible for implementing the fundamental value that these underlying libraries have created.

IPSI

Interactive Physics Simulation Interface, the physics engine produced by Haption to be used for collision detection.

JTOpenToolkit

The library being used to interface with JT files.

VisController

Teamcenter Visualization Controller, the API that controls interaction with Teamcenter Visualization

Teamcenter

Teamcenter Visualization, the tool produced by Siemens PLM Software that will be used to display 3D model representation

2. System Design

2.1. Module Overview

The CATCH library can be broken down into five separate modules: CatContext, CatCursor, CatIPSI, CatJtk, and CatVis. Each module has been designed to hide design decisions from the end user as well as from the other modules. Figure 2.1 shows the high level module interactions.

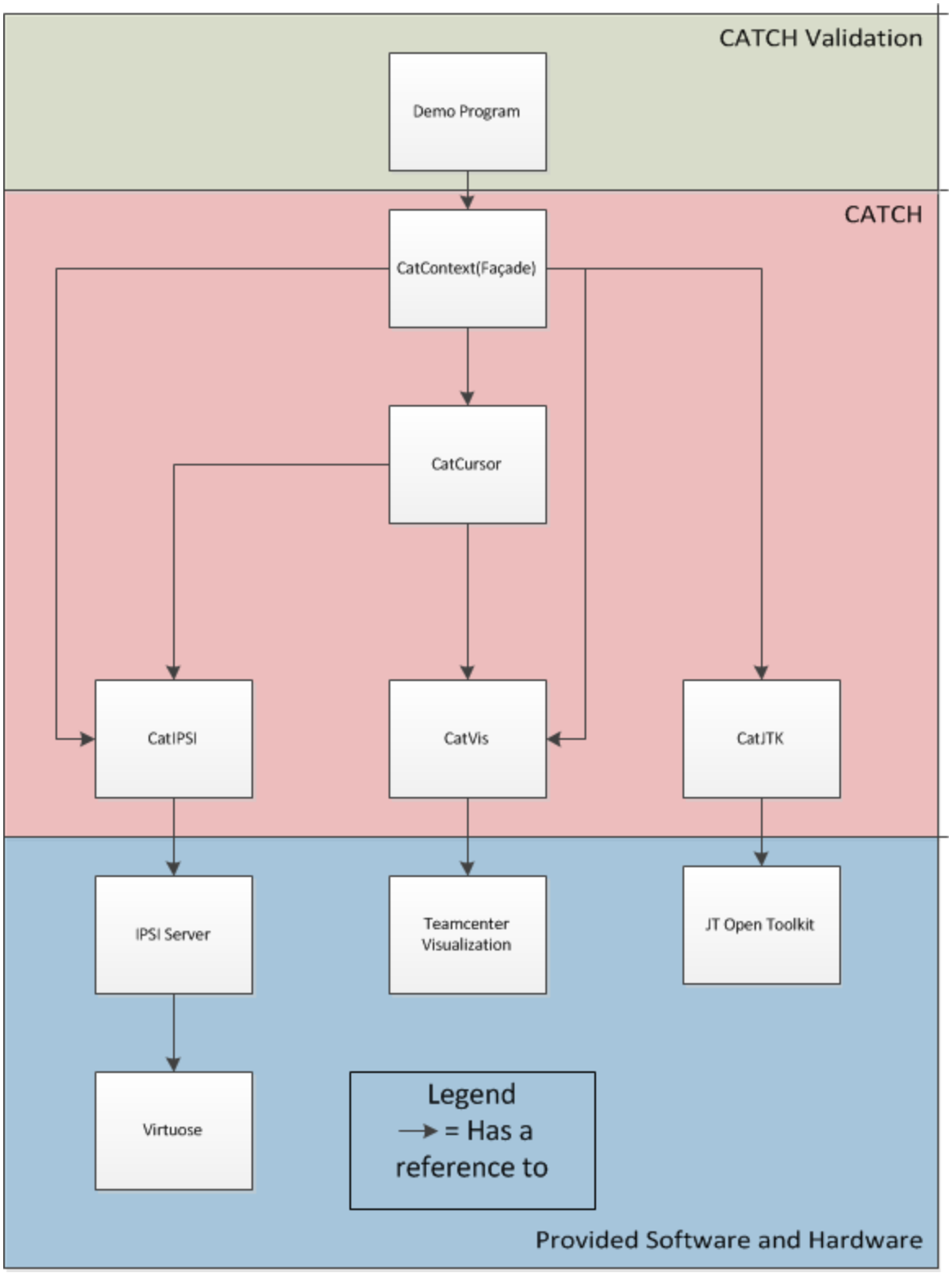
CatContext- This module contains the public facing interface to be used by external users/applications using the CATCH library. This module also contains callback implementation for intermodule communication within the CATCH library.

CatCursor- This module contains the main control loop of the library. It is responsible for all periodic polling for both the manipulation device transformation and model transformations from CatIPSI and transferring them to CatVis.

CatIPSI- This module directly communicates with the IPSI physics engine provided by Haption. The messages sent between IPSI and CatIPSI include all model mesh information, model transformations, and Virtuose state. Note that all haptic feedback and collision calculations are performed entirely by IPSI (provided by our clients) and *not* our application. The API used to communicate with IPSI is also credited to Haption.

CatVis- This module interacts with Teamcenter Visualization using the VisController API.

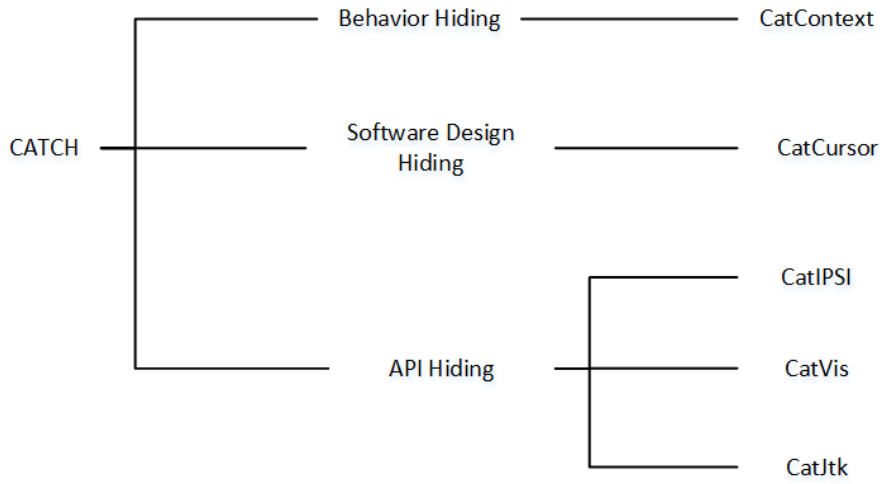
CatJtk- This module is primarily responsible for using the OpenJToolkit library to parse model assemblies, stored as JT files, into individual triangle meshes to be added to the physics engine.



(Figure 2.1) Diagram of the modules of CATCH and their interactions with other modules; and applications, APIs, and devices outside of the CATCH library.

2.2. Module Guide

This section describes the modular structure of the project by breaking down each module into the services, secrets and issues. This guide should help any future parties to better understand our design. A service is defined as the functionality a module provides. A secret is a set of design decisions that are hidden in the module that other modules need no knowledge of the different secrets. Issues are series' of questions that arose during the design of the project that influenced the design of the module.



(Figure 2.2) CATCH module hierarchy with mapping to variabilities and parameters of variation

2.2.1. Behavior Hiding Modules

Behavior hiding modules include any program requiring changes when the output or interaction with the code as a whole is changed. Their secrets relate to the use of the library.

2.2.1.1. CatContext

2.2.1.1.1. Service

CatContext contains the public-facing interface to be used by external users/applications using the CATCH library. It provides high-level methods to initialize the CATCH modules and begin execution. This module also contains callback implementation for intermodule communication within the CATCH library. Additionally, it provides the service of initializing modules.

2.2.1.1.2. Secret

This module contains information about how to use the other modules and the initialization process.

2.2.1.1.3. Issues

Issue: Should CatContext only contain the user interface functions?

(Option 1) Yes, this module is meant to only be a bridge between the CATCH library and its users. Therefore any other functionality should be placed in another module.

(Option 2) No, due to CatContext's position as the facade the initialization process is included as part of the user interface. This means callback functions should be defined here in order to register them elsewhere.

Resolution: Option 1 would truly hide all other functionality from the end user if this user is looking at the top level CATCH source code. Because of the inherent link between the user interface and initialization, other function must be defined here, Option 2. A good example of this are callback functions which must be defined before they can be registered to other modules. Therefore, Option 2 was chosen.

2.2.2. Software Design Hiding Modules

Software design modules hide the software design decisions based on our programming requirements. This generally relates to the interaction between modules and the speed and efficiency of process execution.

2.2.2.1. CatCursor

2.2.2.1.1. Service

CatCursor unifies all modules by facilitating data flow between modules.

2.2.2.1.2. Secret

CatCursor hides the overall program flow by controlling the sequence of data transactions. It also actively governs the execution rate of these transaction, which in turn regulates the speed of CATCH as a whole.

2.2.2.1.3. Issues

Issue: Should CatCursor convert all transformation matrices it sends/receives into a "standard" transformation representation to be used throughout the application?

(Option 1) Yes. Because CatCursor is the means through which the other libraries communicate at runtime, it should be written as an adapter to convert between the different transformation representations.

(Option 2) No. CATCH should take a more modular approach, such that CatCursor should only have to transfer data between modules and not modify it. All modules needing to send/receive transformation information (e.g. CatIPSI, CatVis) are responsible for converting its transformation data to/from a "standard" representation to be passed throughout the application.

Solution: Because we wanted to strive for modularity and consistency throughout our application, Option 2 appeared to be a better solution. We selected our "standard" transformation representation to be that of IPSI, so all modules wishing to send/receive data must first convert that data to IPSI's format.

2.2.3. API Hiding Modules

API hiding modules are the programs that must be changed if an API is going to be replaced by another, for example if a new physics engine is to be used then one of the following modules should be replaced. The below modules' secrets relate to how they interact with their respective APIs.

2.2.3.1. CatIPSI

2.2.3.1.1. Service

Interact with the physics engine: CatIPSI provides the means to simulate the 3D scene. It also provides the means to interact the Virtuose haptic arm and other devices. This provides position and other information about the scene back to the system.

2.2.3.1.2. Secret

This module contains the secrets about how to simulate and manipulate the scene with a haptic arm and the corresponding haptic feedback. In addition, this module is responsible for adding all model vertices to the physics engine in the form of triangle meshes.

2.2.3.1.3. Issues

Issue: Interfacing with the Virtuose, haptic arm. The information about the haptic arm needs to be accessible from both the physics

engine as well as from our main application so that we can render its location to the screen.

(Option 1) Use the Virtuose's API through our application to manually update the location of the device and provide haptic feedback.

(Option 2) Use IPSI's built-in interface to communicate with the Virtuose and poll the device position through IPSI.

Solution: We decided to use IPSI's built in interface because although this would give us less flexibility when interacting with the virtuose, it also greatly simplifies our system and provides an interface for interacting with other generic input devices as well.

2.2.3.2. CatVis

2.2.3.2.1. Service

Display and interacts with the scene. CatVis takes data from other parts of the project and pushes the data to the visualizer in a format that allows the display to update as the scene changes. CatVis must also be capable of receiving information from the visualizer so it can let the rest of the modules know if a part has been selected when a button is pressed. The visualizer used in this project was Teamcenter Visualization, which is developed by Siemens.

2.2.3.2.2. Secret

The secret of this module are the protocols used to interact with the visualizer and the other modules. This module has to be capable of both sending and receiving data to both the visualizer and the other CATCH modules.

2.2.3.3. CatJtk

2.2.3.3.1. Service

Load JT files: CatJtk provides the ability to convert JT files into objects that CatIPSI and the physics engine can use to simulate the scene.

2.2.3.3.2. Secret

CatJtk hides how to read and convert JT files into objects that can be passed into the physics engine scene. It also handles the conversion of JT files into triangle meshes.

2.2.3.3.3. Issues

Issue: Handling primitive shape types. In a JT file, there are two ways to describe an object. One way is to use tri-meshes and the other is to use primitive shape types. For example, a primitive cylinder is defined only by its radius and length, but does not contain the vertices to actually build the shape using a triangle mesh.

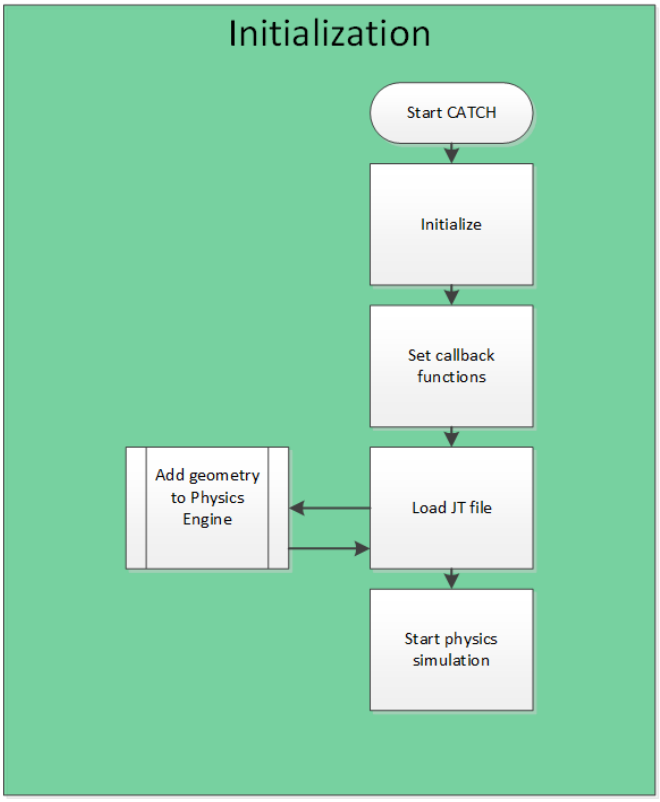
(Option 1) Don't deal with primitive shape constructs. Because of the small scope of the project, a JT assembly without primitive shapes could be selected for our test assembly.

(Option 2) Include the primitive shape usage. Many of the simpler JT assemblies available for use by our demo could contain primitive shapes removing many JT assemblies from consideration.

Solution: We decided not to implement the primitive shape usage. The physics engine in use by CATCH can only have models input by specifying the vertices and indices of a shape. In order to support primitive shapes, we would have to implement conversions for all of the supported primitive shapes in JT Open Toolkit. Due to the scope of our project, we did not want the extra overhead of implementation and testing to support these shapes. Therefore, only JT assemblies made entirely from meshes can be loaded into CATCH at this time.

3. Implementation Details (From a Process Perspective)

3.1. Initialization



(Figure 3.1) Initialization procedure of CATCH

3.1.1. Initialize

Upon initialization, CATCH creates modules, creates an empty physics simulation, checks licenses, waits for VisController client to connect and passes module references to CatCursor. This is achieved by first instantiating the four modules CatIPSI, CatVis, CatJtk, and CatCursor, and then calling each respective init() method on the modules.

3.1.2. Set Callback Functions

Callbacks are registered for adding geometry, processing button changes, and receiving selection state updates. The actual implementations of these callbacks live inside of the CatContext source code, and are set using CatJTK::setTrimeshCallback(), CatIPSI::setButtonStateChangeCallback() and CatVis::setSelectionStateCallback() respectively.

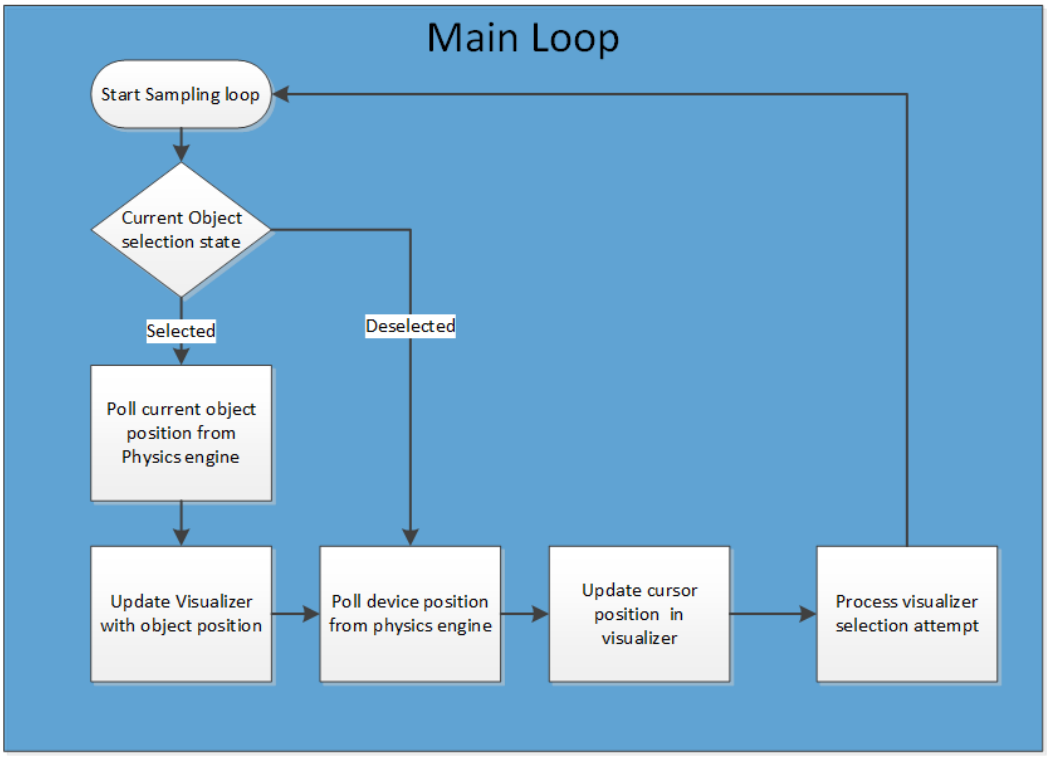
3.1.3. Load JT file

A file containing model meshes is loaded into CATCH by use of the CatJtk module. On each traversed part within the JT file, a triangle mesh will be generated and passed to the callback function set in above step, wherein the callback calls CatIPSI::AddTrimesh() to add the part to the physics engine.

3.1.4. Start Physics Simulation

At this point, all part data has been added to the CatIPSI. Signal to the physics engine to freeze bodies, enable collisions, and set the device baseframe with CatIPSI::startSimulation();

3.2. Main Control Loop



(Figure 3.2) Main control loop procedure of CATCH

3.2.1. Start Sampling Loop

Marks the beginning of the infinite loop that runs when CatCursor::run() is called.

3.2.2. Current Object Selection State

Obtain the current selection state within CatVis, i.e. whether or not a part is currently selected by the cursor, and retrieve the NGID of the part selected.

3.2.3. Poll current object position from physics engine

Using the NGID obtained from CatVis, poll the current part transformation from CatIPSI by use of CatIPSI::poll().

3.2.4. Update visualizer with object position

If a part transformation is obtained for a selected part in the physics engine, the part transformation is immediately updated in the visualizer with the new transformation with CatVis->setTransform().

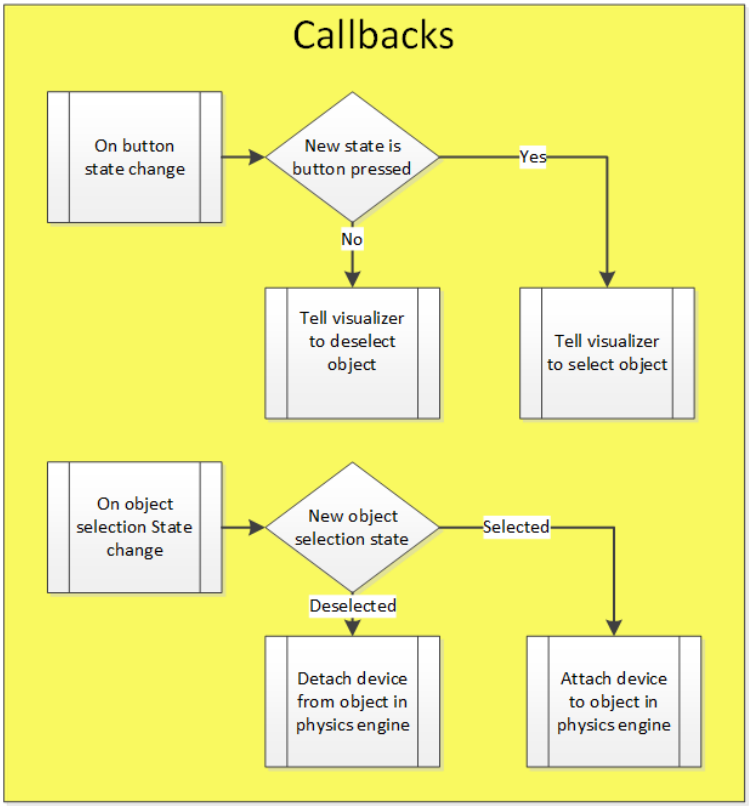
3.2.5. Poll device position from physics engine

Obtain the current device transformation from the physics engine by use of CatIPSI::pollDevice().

3.2.6. Update cursor position in visualizer

Update the cursor transformation in the visualizer with the device transformation using CatVis>setCursorTransform().

3.3. Callbacks



(Figure 3.3) Part selection callback procedure

3.3.1. On button state change

This callback is registered with CatIPSI. Whenever CatIPSI polls the device position it also checks the button state of Virtuose by use of the IPSI API. The callback function itself lives in the CatContext source code.

3.3.1.1. New state is button pressed

The behavior of the buttons on the Virtuose in CATCH is based on whether or not the button is currently being held down. Pressing the button triggers a selection attempt. To remain selected to an object, the user must keep holding the button down. As soon as the user lets go of the button, the part is deselected. In short, the selection state in the visualizer is driven by the button state of the device.

3.3.1.2. Tell visualizer to deselect object

When the new button state is *not pressed*, CatVis uses the VisController API to deselect the currently selected part with CatVis::deselectAllParts().

3.3.1.3. Tell visualizer to select object

When the new button state is *pressed,* CATCH attempts to select a part using CatVis::attemptSelect(). A part will only be selected if the cursor is currently touching it when the button is pressed.

3.3.2. On Object selection state change

This callback is registered with CatVis. For every part action that happens in Teamcenter Visualization, the VisController API calls an inner callback in CatVis that notifies CATCH whenever a part has been selected. CatVis exposes an outer callback to CatContext to trigger certain events when a part is selected in Teamcenter.

3.3.2.1. New object selection state

The behavior when an object changes selection state is fairly straight forward. Essentially, the visualizer drives the attachment state of parts to the representation of the Virtuose within the physics engine. When a part is attached to the device, the part tries to transform along with the device.

3.3.2.2. Detach device from object in physics engine

Detaching the device from whatever object it is currently attached to in the physics engine is accomplished with CatIPSI::detachDevice().

3.3.2.3. Attach device to object in physics engine

Attaching the device to an object within the physics engine requires use of an NGID to specify which part to attach to. This NGID is one of the things polled for by CatCursor within the main control loop. Attaching is accomplished with the method CatIPSI::attachDevice(std::string oid).

4. Module Design Rationale

This section describes the design decisions and rationale behind how the main service of each module was implemented.

4.1. CatContext

4.1.1. User-accessible Functionality

CatContext was designed to be the facade module of CATCH. It handles interaction with users of the library and registers all intermodule callbacks. Because this is the external-facing module, we wanted to provide cleaner, high-level methods of initializing and running the CATCH library. To fully utilize the CATCH library, only three functions need to be called: CatContext::loadJtFile(), CatContext::init(), and CatContext::run().

4.1.2. Intermodule Communications Callbacks

Because of our focus on a modular approach in designing CATCH, we chose to handle much of the inter-module communication by using callbacks to allow data to pass between modules without requiring these modules to know about one another. By registering inter-module callbacks during initialization, each module is more easily replaceable, if needed.

There are three callback functions contained within the CatContext module. These are hidden from the user, as they are not included in the CatContext header file:

callback__jtGeometryToIPSI(): This callback is responsible for transferring geometry information from CatJtk to CatIPSI. During initialization, it is registered to CatJtk to be called at runtime. When initiated, the provided geometry mesh information is passed to CatIPSI by calling CatIPSI::addTrimesh().

callback__buttonStateChange(): This callback is responsible for communicating haptic device button state changes from CatIPSI to CatVis, and is part of the part selection process. It is registered to CatIPSI during the initialization process. Whenever IPSI detects that a button is pressed / release, this callback is triggered and CatVis is notified of the change by calling CatVis::attemptPartSelection() or CatVis::deselectAllParts(), respectively.

callback__partSelectionStateChange(): This callback sends any changes in part selection state and, if necessary, the ID of a new object that should be attached to the haptic device in the physics engine. This callback is

registered to CatVis during the initialization process. It is triggered after a part selection attempt or part deselection occurs. If an object is selected or deselected, CatIPSI is instructed to either attach or detach the manipulation device (i.e. the Virtuose) to the object by calling CatIPSI::attachDevice() or CatIPSI::detachDevice(), respectively.

4.2. CatCursor

CatCursor was designed to be the most "centralized" module in the CATCH project. Its initial purpose was to keep track of the cursor position, as well as any other state management required by the application. Later, as the CATCH design began to solidify, it became apparent that there is little need for state management in CATCH, as most necessary state is stored by IPSI and Teamcenter. Because of the desire for CATCH to follow more of a "plug-and-play" design, (e.g., a new physics engine could potentially be introduced and a CatIPSI could be replaced by a new interface), it was decided that CatCursor should not be involved with the implementation details of the API-facing modules. Instead, CatCursor's role is to simply transport data between modules; it does not perform any operations on the data itself. Because the APIs used by CATCH require polling to get new data, CatCursor is in charge of the following three duties:

A. Requesting data from our main API facing modules (CatVis and CatIPSI)
B. Sending polled data to the module that requires it
C. Controlling the rate of execution

CatCursor only has one primary method: *CatCursor::run()*. This function launches a while loop that begins by sleeping the active thread by an amount of time that will result in the loop executing at 30 iterations/sec. This sleep time is determined by the following formula:

$$T_{sleep} = T_{expexcted} - \frac{Clk_{current} - Clk_{previous}}{f_{clk}}$$

where $T_{expected}$ is the expected execution frequency of the loop (1/30th of a second, or 33.3ms), $Clk_{current}$ is the starting clock cycle of this iteration, $Clk_{previous}$ is the starting clock cycle of the previous iteration, and f_{clk} is the clock frequency of the application. After this sleep, CatVis is polled for the currently selected part. If a part is selected, CatCursor requests the transformation (i.e., position and orientation information) of the corresponding body in the physics simulation from CatIPSI, and forwards that data to CatVis to update the 3D rendering. Similarly, CatCursor then polls for the transformation of the manipulation device (Virtuose) from CatIPSI and sends that to CatVis to be rendered as the cursor. CatCursor deals very little with how those transformations are represented, allowing changes in the data representation without changes to CatCursor.

4.3. CatIPSI

CatIPSI was designed to act as a generalized interface between the IPSI API and CatCursor. The Intent is for the Interface to be general enough to be able change physics engines with minimal API changes.

When we add bodies to IPSI we add a separate triangle mesh for each body in the simulation. A triangle mesh is a list of vertices, and list of indices that describe how the vertices are connected into triangles. Then we specified an overall transformation for the completed object. These are received from CatJTK through a call back, and added to IPSI through the IPSI API. when we add each of thes files we also give the Cat IPSI module the NGID from the JT file. We use a map to translate between the NGIDs and the bodyIDs that are internal to IPSI for the object. This way IPSI's Ids are segregated to only the module that directly interacts with IPSI and they are hidden from the rest of the program.

When we need to poll the position of a body in IPSI we pass the NGID of the part into CatIPSI and using the map that was mentioned earlier we translate it into the body id. We then poll the position of that specific body in the simulation.

When we need to poll the position of the manipulation device we just call the ManipulationDeviceGetPosition() method and return the transformation matrix to the calling module.

Whenever the Device position is polled the Current button state of the device also polled. This checks if the button state has changed and if the button was pushed it triggers a callback to attempt to attach an object to the device through CatVis. If the button is released then the body is detached from the arm. Deselection occurs when the selection is attempted and there is no object within the cursor. It is implemented this way because VisControler does not currently support the deselection of parts via callbacks.

Simulation step size is set during the initialization of CatIPSI and cannot be changed at run time.

4.4. CatVis

Similar to the decision behind creating CatIPSI to hide physics engine interaction details, CatVis generically represents a 3D visualizer; it acts as a generalized interface between CatCursor and the VisController API. It provides methods for performing operations such as selecting or deselecting a part, sending updated

VisController utilizes a number of callback interfaces to notify subscribed listeners about relevant events that have occurred. By registering callback functions with

VisController, CatVis is able to track relevant Teamcenter state information, such as the position of the camera within the scene (by receiving the view transform), or the currently selected part(s) within the scene.

The view transform retrieved through the Viscontroller View Callback is needed to convert the cursor position back into world coordinates when the camera view changes. When cursor coordinates are sent through Viscontroller, they are applied to the cursor in the frame of the camera. Without accounting for the view transform, the cursor would remain in the same position relative to the screen when the camera moves. This behavior does not work since it causes the Teamcenter cursor location to become out of sync with the IPSI device transformation.

When CatVis::AttemptSelection() is called on CatVis, it uses the VisController API to perform what is equivalent to a click in Teamcenter. Typically the user has alreadyThis action is asynchronously performed, and the result is obtained through the PartActionCallback registered with VisController. References to that part are saved within the CatVis state. External classes can access the selected part's NGID, but they cannot retrieve any of the VisController API specific information.

To modify the selected part's transformation within the Teamcenter visualization, the public CatVis::setPartTransform() method is used. The input transform, which is part rotation matrix and part translation is converted to a full row major 4x4 transformation matrix and set to Teamcenter using the VisController API function sendPartAction().

In CatVis one limitation is that there isn't a method to set a part's transformation if that part isn't already selected. When trying to implement this functionality we ran into difficulties crafting a VcPartData object from scratch that VisController would accept. This issue prompts further investigation and may be a chance for further work. Transforming a selected part works well since the VcPartData object is created from within the VisController API.

Deselecting a part in CatVis represents a difficult challenge when using the VisController API. Even though a SendPartAction(DeselectAll) type function exists in the VisController API, it is not currently implemented . To overcome this limitation we have to perform an action similar to moving the cursor away from the part and clicking twice on the background. CatVis combines the functionality of attemptSelect() and setCursorTransform() methods to perform a deselect.This happens in less than 100 ms and sometimes timing issues occur when multiple clicks are not registered due to their short time interval.

4.5. CatJtk

4.5.1. Implementation Details

The main service that CatJTK provides is to load a JT file, convert part representations into triangle meshes, and expose those triangle meshes to be loaded into the physics engine. The design rationale for CatJTK is mainly a discussion of the implementation of the method CatJtk::loadJTFile(std::string filepath).

CatJTK uses the OpenJTToolkit API to do most of the heavy lifting. OpenJtToolkit traverses the JT hierarchy, calling the CatJTK implemented callback of type JtkTraversActionCB for each part that is traversed. Therefore this is really only a discussion of how the traversal Callback is implemented.

The callback procedure for each node of the JT file is as follows:

If the node is of type JtkPART the corresponding NGID is extracted as the path to the node. The global transform for the part is calculated by looping through each parent multiplying matrix transformations until the root node is reached. Polygons within the part are then added to a JtkTriangleStrip. That triangle strip is packaged within a CatTrimeshJtk object and exposed to the rest of the CATCH library through the TrimeshCallback. CatTrimeshJtk exposes the vertices and indices of the triangle mesh representing a part within the assembly.

if the node is of type JtkINSTANCE, the node is used for calculating NGID and global transform but is not directly used to access polygonal information. Instead the original part reference is acquired and used in a similar way as the JtkPART case.

CatJTK currently works on only a subset of possible JT file configurations. Files cannot contain nested JtkASSEMBLY types, as it is assumed that the file is made entirely out of parts, and instances, in a single assembly. Secondly, CatJTK only operates on files made up entirely of parts expressed as polygons. It does not work with files that use primitive data types such as spheres or cylinders to express shape information.

In order to avoid maintaining an internal memory store with CatJTK, the callback pattern is used to allow part information to be accessed as soon as it has been parsed from the JT file. The tradeoff being that if external modules wish to store that data in the long run, they would be responsible for allocating their own memory for it. This pattern works well in the case of CATCH since the IPSI

physics engine provided by Haption already tracks its own mesh data within the simulation. By using the callback pattern, memory allocation is kept at a minimum while still allowing a high degree of modularity.

5. System-Level Technical Challenges

5.1. Transformation Representations

VisController & OpenJTToolkit
Matrix Representation

As double[16]
Each number represents it's index in the double array

Rewritten to match the form of a classical 4x4 array

0	1	2	12
3	4	5	13
6	7	8	14
9	10	11	15

Indices [12], [13], [14] are the x, y, and z translation

Indices [0] – [8] represent a row major 3x3 rotation matrix

Indices [9], [10], [11] are typically 0, and [15] is typically 1

IPSI (Physics Engine)
Matrix Representation

As double[16]
Each number represents it's index in the double array

Rewritten to match the form of a classical 4x4 array

0	3	6	12
1	4	7	13
2	5	8	14
9	10	11	15

Indices [12], [13], [14] are the x, y, and z translation

Indices [0] – [8] represent a column major 3x3 rotation matrix

Indices [9], [10], [11] are typically 0, and [15] is typically 1

VisController, OpenJTToolkit, and IPSI
Quaternion Representation

As double[4]
Stored as scalar last

Q = a + bi + cj + dk

Rewritten to match the form of
x y z w quaternion terms

quat[0] = X = b
quat[1] = Y = c
quat[2] = Z = d
quat[3] = W = a

IPSI double[7] Transformation Representation #2

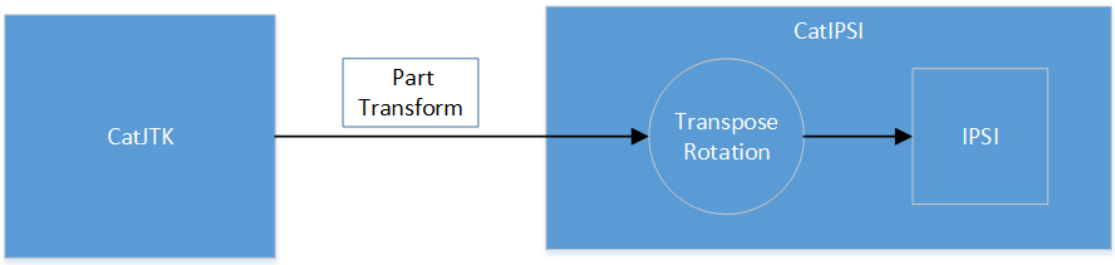
Array is split into Position and Orientation

double[7] M
Position:
M[0] = x coord
M[1] = y coord
M[2] = z coord

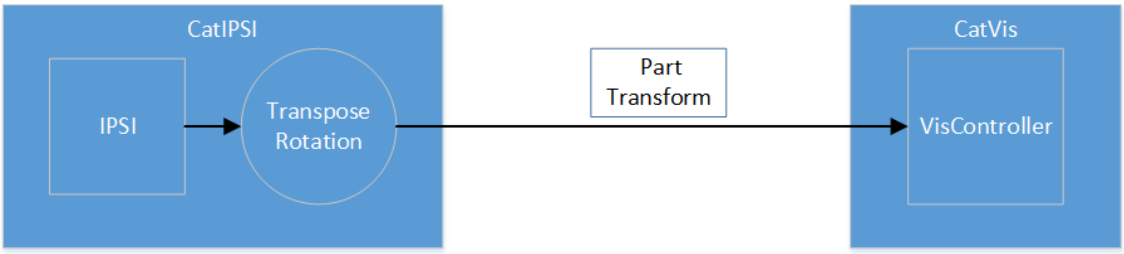
Quaternion:
Q = a + bi + cj + dk
M[3] = X = b
M[4] = Y = c
M[5] = Z = d
M[6] = W = a

5.2. Transformation Manipulation

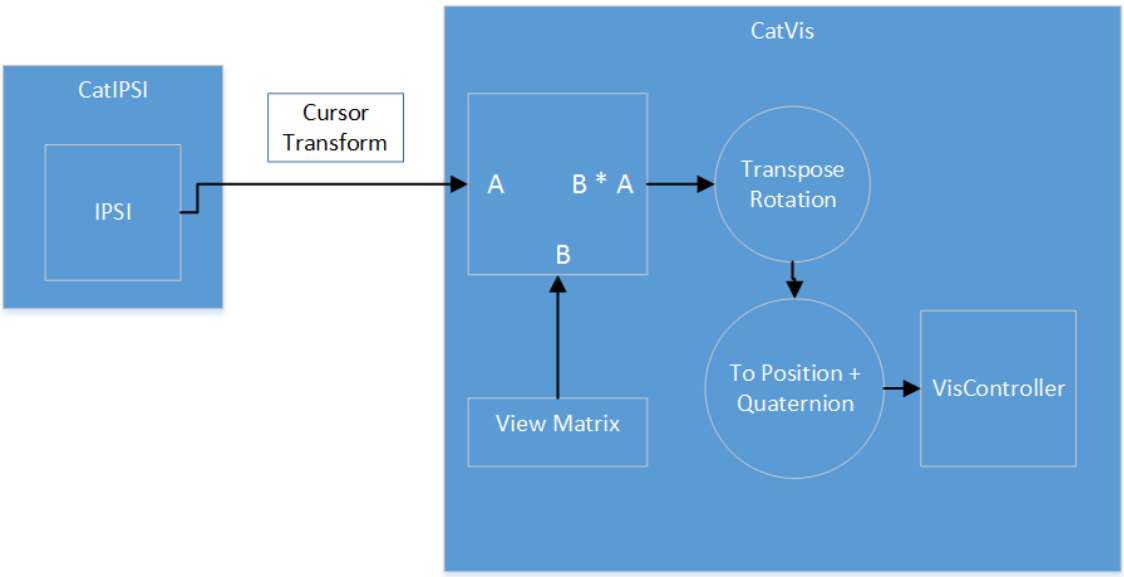
Part transformations from JT file to IPSI physics engine



Part transformations from IPSI physics engine to VisController API for Teamcenter Visualization



Cursor transformations from IPSI physics engine to VisController API for Teamcenter Visualization



5.3. Memory management in C++

Because this application was the first large C++ project for most of our team, we encountered a number of problems related to memory management practices in C++. Many of the module interfaces were designed with data scope and lifecycle in mind. Generally speaking, we opted to have each module allocate and "own" any data needed for that module's operation for the duration of its lifecycle. If that data is required to be updated by an external module, pointers or references to that data are passed to that module so that the original location in memory can be updated. Although this seems to be a good practice for ensuring that relevant data remains in scope for the entire duration of execution, we still encountered many confounding problems in memory management when passing data references to our project dependencies (e.g. IPSI or VisController). For example, because of the many ways that strings can be represented in our application (including C-style strings, std::strings, and vector<byte> objects), it often became difficult to determine where that memory was allocated, and who "owned" that data and its respective lifecycle. Much of these problems were avoided by eventually allocating buffer memory on the heap to store data with confusing or unknown owners, and ensuring one module owned that newly allocated heap data.

6. Testing

6.1. Testing Procedure

The testing protocol for CATCH was determined by our client specifications and our meetings with our client. Very early on in the project our client made it clear they were only looking for an application that will be used as a proof of concept to demonstrate specific functionality within a limited use case. Therefore no strong testing requirements were placed on the project. With this in mind it was decided to prototype and demo the current state of the project at bi-weekly client meetings. For these incremental demos we used devices other than the Virtuose haptic arm. In its place a six degree of freedom mouse or the Virtuose Simulator provided by Haption was used as the manipulation device for many of our demos. Once we had confidently achieved minimum functionality using the simulator and/or space mouse, we began testing with the Virtuose itself in METaL. Testing with the Virtuose was very helpful for debugging issues with the physics engine and visualization scene. It also helped the optimization of haptic feedback.

6.2. Results

The main software deliverable of first semester was to get our application to move the cursor in Teamcenter visualization. We were successful in doing this but the transforms for the cursor were not correct. Next we focused on correcting these transformations and correctly adding Bodies to IPSI. Once the ability to manipulate parts in Teamcenter Visualization through VisController was added we found that the transformations of our parts and cursor did not match. We figured out that the way the transformation matrices

were being stored in memory was different in both VisController and IPSI. It took a while to figure out where we needed to transpose our matrices and how to convert between them to get everything working correctly. This was aided by discoveries we made when trying to attach the device to a body at an offset. After that was figured out we were able to correctly transform the cursor and bodies while applying responsive amounts of haptic feedback to the Virtuose. So in the end we were able to load parts from JT files in to our physics engine. Transform them in Teamcenter Visualization using the VisControler API. Then we were able to manipulate them using the haptic device while providing the user with appropriate haptic feedback.

6.3. Outcome

The CATCH project was able successfully demonstrate current functionality at our client meetings throughout the year. We were able to incrementally extend functionality so that we had something new to demo almost every two weeks. We ended up with a prototype that successfully met all of the clients' requirements and exceeded their expectations.

7. Reflection

This project experience has taught our group many things about working with haptic technology, physics engines, visualization software and the issues associated with integrate multiple APIs especially when they all are attempting to represent the same thing. One of our biggest learning experiences dealt with memory allocation in C++. We know it would be an issue at the start of the project and we did find an acceptable solution but as we reached the end of the project we realized there were many things that we could have done better in this respect.

One of our biggest issues our project encounter was dealing with transformation matrices. We ran into issues with standard matrix transforms and quaternion representation. We ran into this issue when dealing with the device position in the physics engine. We would retrieve the device position for the cursor and read the device base frame in quaternions, but when we needed to set the device base frame the physics engine expected matrix representation. This was very confusing because we expected to set the position in the same form that we retrieve it. The other major issue we had with transformation matrices was the different way different APIs represented the same transformation in different ways. It took a long time to map the transformation matrices between the physics engine and VisController, this was a huge issue that blocked our progress for a large period.

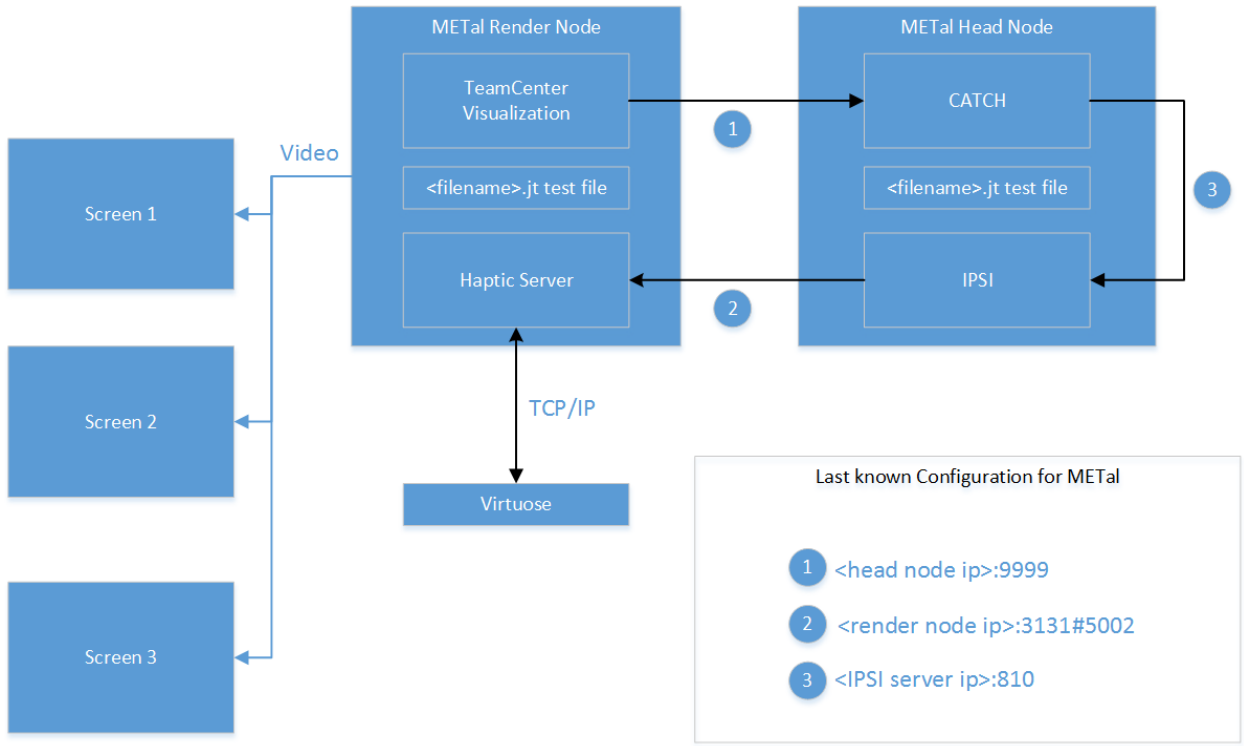
The physics engine we are using does block the ability to complete some basic tasks. One issue is the voxel resolution settings. This affects the ability to put a cylinder into a round hole. The physics engine is interpolating all of the triangle mesh objects a sets of cubes. This will cause the false collision that are not actually occurring. This will occur due to the relatively large voxel resolution setting of 2mm needed for optimal haptic feedback from the device. The physics engine also eats up large amounts of memory. This is why CATCH was set up to interface with the physics engine over a network. When we tried to load an average JT assembly with an average number of part and and the physics simulation was wanted to use 16GB of memory.

This is a major issue with the scalability of this project that would need to be agresed by the API. This is probably meant to be used with large worlds where there are things like desks and chairs not tightly bound parts that are meant to be disassembled.

The VisController API is still in development and could be improve in order to make the haptic interface better. Our one major issues involves the way parts are deselected. There should be a way to deselect parts without having to click on empty space. From a user perspective it makes much more sense to hold down a button on the device to select a part and the release the button to release the part. Clicking on empty space in order to deselect an object makes sense for a standard visualization software but doesn't work well in immersive environments used with haptic devices.

While CATCH has successfully proved the viability of integrating haptics into commercial visualization software there are definite improvements that would improve CATCH. Currently CATCH is really only useful as a standalone application currently. The CatCursor module would need to be modified in order to make CATCH truly just a piece of larger whole. CatJtk also needs to be modified to work with all types of JT files. This includes the addition of support for JT assemblies that contain sub-assemblies and the ability to convert primitive geometry types into triangle meshes. It is also important to add change the view in the visualizer based on the position of the users head. This would increase the immersion of the user. This involves both updating the view matrix in the visualizer and the device base frame in the physics engine. This is something that could be test using the head tracking system already available in METaL. It would also be nice to add the ability to select multiple parts at one time. This could be useful if there are multiple devices connected to the same simulation scene or if a large number of small parts need to be relocated at one time. Finally, it is imperative so solve the voxel resolution and memory issues we encountered with the physics engine. This might have to be resolved by switching to a different physics engine or working with Haption, the designers of the physics engine to have more options for this type of application.

A.1 Appendix I: Operations Manual



(Figure A.1) Layout of METal and location of resources on Head and Render node

Running CatContextTest Demo in METaL: Step-by-Step Instructions

Build Instructions

1. Do a 'git clone' of the CATCH repository, if needed
2. Open up the 'CatContext' solution in Visual Studio 2010
   *Note: Visual Studio 2010 SP1 is required
3. Set the project to be built in 'Release' and 'x64' mode at the top of the screen
4. Set 'CatContextTest' as the StartUp Project
5. Build the CatContext solution (*not* the CatContext project)

Configuration

0. METaL lab setup/Virtuose setup

Resources and instructions for METaL projectors/Virtuose are available through VRAC resources and will not be included in these instructions. These are available to authorized users at:

https://intranet.vrac.iastate.edu/twiki/bin/view/TheLab/VanceGroup

Users must be trained on METaL lab procedures prior to executing this demo.

1. Head Node: Set IPSI IP address

In CatContextTest.cpp, verify that the correct IPSI IP address is set to be used. This is the IP address that the project uses to access the physics simulation during initialization. In this case, IPSI is running on the same machine (the head node), so the IP address should be 127.0.0.1.

2. Head Node: Configure IPSI to use Virtuose Haptic Server

Open the “Device Configurator” for IPSI (shown in Figure A.2), available at:

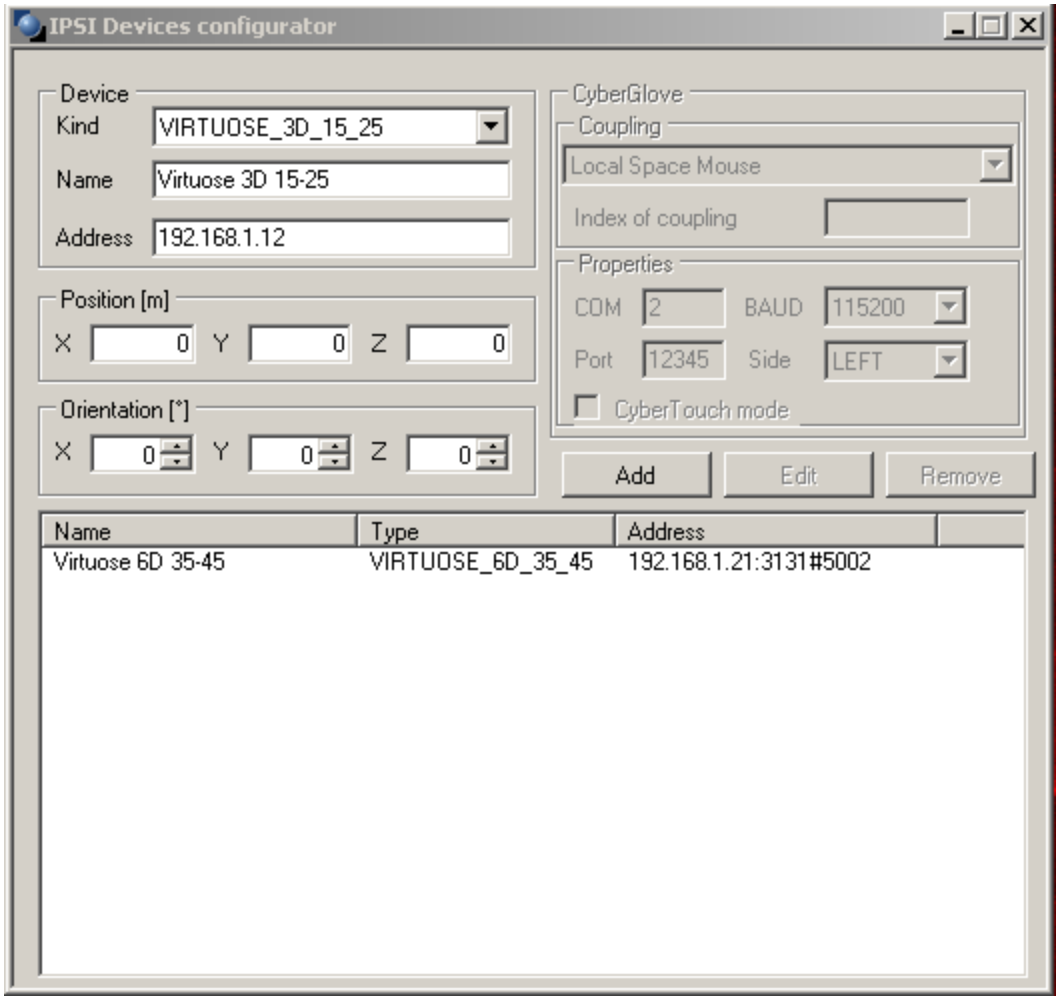
C:\Program Files\HAPTION\IPSI\Server\V2.10\bin

If not already present, add a “Virtuose 6D 35-45” device to the list of configured devices. Set the Address field to be the IP address of the of the render node, the local (head node) port to be used, and the remote (render node) in the following format:

<RemoteIP>:<LocalPort>#<RemotePort>

At the time of writing, this IP address is:

192.168.1.21:3131#5002



(Figure A.2) Correct "Device Configurator" configuration

3.Render Node: Configure Teamcenter

Teamcenter needs to be configured with the correct VCD, SCD, and ImmersiveConfig files. These files can be found at:

\doc\TcVis_config_files\METaL_Configuration

Copy these files to C:\Temp before using.

To load configuration files for CATCH using METaL projectors, open Teamcenter and navigate to File->Preferences->Immersive Display->Configuration, and select the correct ImmersiveConfig file.

*Note: in the ImmersiveConfig file, the path to the head node must be correctly specified. At the time of writing, this value is:

```
<VisController>
   <Server name="192.168.1.20:9999"/>
   ...
```

</VisController>

In the VisController API, Teamcenter serves as the client, where CATCH acts as the server (by using the VisController.dll) Additionally, up-to-date VisController DLLs must be on the machine to support the functionality that CATCH uses.

4. Render Node & Head Node JT file setup:

Put the JT model file that you wish to use for the demo at locations that are available from both machines. It can be two separate files as long as they are identical. The JT file must be one that doesn't contain nested assemblies, and only contains geometry in the form of polygon meshes. The two files recommended for use are

\Catch\CatJtk\CatJtkTest\test_shapes\asm_no_scale.jt
\Catch\CatJtk\CatJtkTest\test_shapes\garage_door_opener_reduced.jt

This garage_door_opener file has some parts removed for usability. The small gear is the only part that can be moved due to the generated Voxel maps being very tightly bound.

5.Head Node: Tell CatContextTest to use the correct JT file:

There are currently two ways of specifying which JT file to load with CatchContextTest:.

Option 1: define the JT_FILEPATH macro at the top of CatchContextTest.cpp

Usage:

```
#define JT_FILEPATH  <File Path>
```

Example:

```
#define JT_FILEPATH
        "C:\\siemens_tcvis_haptic_13-14\\Catch\\
        CatJtk\\CatJtkTest\\test_shapes\\asm_no_scale.jt"
```

Option 2: Specify a file argument when starting CatContextTest

Usage:

```
CatContextTest <File Path>
```

Example (executed via command line):

```
CatContextTest
"C:\\siemens_tcvis_haptic_13-14\\Catch\\
CatJtk\\CatJtkTest\\test_shapes\\asm_no_scale.jt"
(All as one line)
```

Starting the Demo

Assuming that METaL projectors are turned on (as per the METaL user guide) and the Virtuose is ON and connected…

1. (Head Node) Verify that no IPSI console windows are still open on the machine running IPSI server. Teamcenter should be off.

2. (Head Node) Start CatContextTest and wait for the “Waiting for VisController to connect” message.

   *Note: The CatContext solution builds all executables to the folder:
   \Catch\CatContext\x64\Release

   Once the demo is started, the info messages displayed in the CatContextTest console window should indicate whether or not CATCH connected to the Virtuose successfully.

3. (Render node) Start Teamcenter.

4. (Render node) Open the JT file to be used for the demo.

5. (Render node) Click the checkbox in the model viewer so that it is visible in the visualizer

6. (Render node) Start Immersive mode in Teamcenter by either typing Alt + C + I + A or clicking Concept->Immersive Mode->Activate . At this point, Teamcenter should connect to the CATCH demo program running on the head node. If it doesn’t, check the immersive mode configuration and verify the correct IP address for the head node is entered as the VisController Server.

   *Note: Each time Teamcenter enters Immersive mode and is then deactivated, the Teamcenter application must be restarted before being able to reconnect to CATCH.

7. Wait for the JT model to load in CATCH, until the message “IPSI Simulation Started” appears in the console.

8. The demo has now been started and you should be able to manipulate the immersive cursor with the Virtuose and see it on the projectors. Note that it is important that the view is not manually rotated within Teamcenter. This is because the cursor is always multiplied by the inverse view transform. If the view is rotated, moving forward with the virtuose will move the cursor in a direction that does not feel like forward with respect to the viewer. Instead the cursor always moves forward with respect to the global frame.

9. When restarting the demo, remember to close any IPSI console windows that are keeping the simulation alive. If the demo is restarted without doing this, it will fail with an RPC error.

Using the Virtuose in the Demo:

1. Push the middle red button in order to lock the cursor in place and readjust the virtuose to any position. This is useful for when the cursor starts too far away from the model and must be moved closer in a series of translations.
2. To select a part, move the cursor over a part until it is highlighted in Teamcenter. Press and hold the left button.
3. While a part is selected, move the Virtuose to move the part. Appropriate haptic feedback will be produced if the selected part collides with another part in the assembly.
4. To deselect a part, let go of the left button on the Virtuose.

Using CATCH Library:

Required Libraries

CatContext.lib
CatCursor.lib
CatIPSI.lib
CatVis.lib
CatJTK.lib

Includes:

```
#include "memalloc.h"
      *Note: This is required for RPC
#include "CatContext.h"
```

Code:

This section demonstrates the simplicity of using the CATCH library from a coding perspective. Only four functions need be called to setup and run the CATCH library:

```
Catch::CatContext* _catContext = new Catch::CatContext(IPSI_IP);
_catContext->init(SIMULATION_MAX_OBJECTS,
      IPSI_STEP_TIME, IPSI_RESOLUTION);
_catContext->loadJtFile(JT_FILEPATH);
_catContext->run();
```

Demo project:

If you have access to the CATCH repository, open up the CatContext solution to have a fully set up solution with correct relative paths to dependencies:

\Catch\CatContext\CatContext.sln

Note: this project only works in x64 Release build mode due to dependency limitations.

Debugging Notes:

Debug Output

Throughout the source code you will find commented blocks of print statements that can be used to display useful debug output, such as transformation matrix values.

Using a Space Mouse

By default, the project is set up to use the Virtuose only, but during our development we found that it was useful to also be able to use a 3D space mouse. In order to use the space mouse, uncomment the line found in \Catch\shared\CatUtil.h:

//#define USE_SPACE_MOUSE

When the space mouse is enabled, only the space mouse may be used as input (not the Virtuose). Similarly, without the space mouse enabled, only the Virtuose can be used.

Additionally, TeamCenter can be configured to run on a single desktop monitor while in immersive mode, instead of a 3-sided CAVE environment. Configuration files for doing this can be found in \doc\TcVis_config_files

Using the Virtuose Simulator

1. Build the project *without* USE_SPACE_MOUSE defined
2. Launch the Virtuose Simulator application (do NOT push the 'Power On' button)
3. Launch the CatContextTest application. Wait for the application to indicate that it is waiting for a VisController connection.
4. Push the 'Power On' button on in the Virtuose Simulator window.
5. Launch Teamcenter Visualization (TcVis).
6. In TcVis, open the JT file specified in CatContextTest.cpp.
7. In TcVis, enter Immersive Mode by navigating to:
   Concept>Immersive Display>Activate (or press Alt+C+I+A)
8. The application is now set up. The simulator can be used to manipulate the cursor in TcVis.

About Licensing:

Teamcenter Visualization Mockup, IPSI (physics engine), and JT Open Toolkit all require licences for use. The IPSI licences must be pointed to manually when a program using the API is run on a computer for the first time. At VRAC, this licence will likely be located in the root of the C:\ drive with the computer's MAC address as the licence file name and a .lic extension.

A.2 Appendix II: Alternatives

A.2.1. Virtuose API

Before we knew that IPSI had native support for the Virtuose as well as several different types of input devices, we considered having the Virtuose API contained in its own module.This module would have been responsible for polling device position and sending haptic feedback forces back to the device. After we realized that this was not a necessity we decided to make a more generalized module that contained both the haptic device and the physics engine, enabling us to make use of this built in native support for the device.

A.2.2. Visualizer at the Center

Initially when we talked with our client about the project the Siemens it sounded like the project was supposed to be centered around the visualization software. CATCH would update the physics scene with the position of the objects. Then the physics engine would detect collisions and report back any haptic feedback feedback forces. After investigating the physics engine we discovered that it made more sense to update the visualizer with data from the physics scene. The physics engine could already represent devices and use the input from a device to transform objects. This original idea assumed that the physics scene couldn't represent a device, once this idea was shown to be incorrect we talked to our client about changing the design specifications.

A.2.3. Using the Physics Engine for Part Selection

When we reach the end of the first semester, VisController didn't have have the ability to send part selection information yet and we were a little nervous. We discussed how to mitigate this issue that could have seriously blocked our project from being completed. We discussed selecting parts through the physics engine and then updating the visualizer with the part data. We decided against this because our client completed this feature of VisController early on this semester. This ended our ability to consider this implementation because our client wanted us to test this feature so proposing an alternative when their API was ready was not going to be an option.

A.3 Appendix III: Other Considerations

The chosen testing approach caused a major unexpected consequence with our design process. In mid to late February when CATCH was almost ready for testing with the Virtuose, Dr. Vance's research group was having issues with the Virtuose functioning properly. It was later found that the device had multiple malfunctioning boards that required the entire device to be shipped to Haption (in France) for repairs. The haptic device was gone from March 13th, 2014 until April 17th, 2014. This significantly reduced the amount of testing time available with the final configuration of the project. Because of this, we were unable to implement some additional features the library due to the additional time spent debugging the library with weaker testing parameters.

The next risk was caused by VisController API. Siemens is currently developing this API so there was no way to know when features of this API would be ready and available to our development team. This was mitigated by our iterative implementation and testing strategy.

Another risk that persisted throughout development was our uncertainty in the way each library formatted its representation of a 3D transformation. Each library we interacted with has its own way of representing a transformation matrix. Some libraries also employed the use of position/quaternion combinations to represent 3D transformations.

A.4 Appendix IV: Definition of Terms and Acronyms

Term / Acronym	Definition
Callback	A callback is a mechanism by which a routine can be set to be executed from within an underlying class. A callback is essentially a function pointer that is called by an inner class to to signal an important event. When the callback is executed, user defined code can handle the event along with accompanying data.
CATCH	Collision Detection and Teamcenter Haptics
IPSI	Interactive Physics Simulation Interface, the physics engine produced by Haption to be used for collision detection.
JT	A data format for 3D models developed by Siemens PLM Software
Jttk	JT Open Toolkit, the software being used to interface with JT files.
NGID	New Genereration Identifier is a construct used by the VisController API to refer to specific parts within a JT file assembly. The term “OID” or “Object ID” may refer to this as well.
Space Mouse	A term used to describe a handheld device that can be used to manipulate a 3D scene with 6 degrees of freedom (X, Y, Z, Yaw, Pitch, Roll).
TCVis	Teamcenter Visualization, the tool produced by Siemens PLM Software that will be used to display 3D model representations.
Transformation	Translation, Rotation, and Scaling of vertices
Triangle Mesh	A triangle mesh is a set of triangles that describe a three-dimensional surface.
VisController	Teamcenter Visualization Controller, the API provided by Siemens PLM that controls interaction with TCVis.
Virtuose	A haptic arm produced by Haption that is capable of manipulating a 3D scene and providing haptic feedback.