Computer Engineering, Degree Project, Second Level, 30 higher education credits

A SIMULATION TOOL FOR HUMAN ACTIVITY RECOGNITION

Erik Westholm

The Simulation and Game Design Master Program 240 credits

Örebro, Sweden, Spring 2010

Examiner: Federico Pecora
Abstract

The goal of this project was to create a simulator that was to produce data for research in the field of activity recognition. The simulator was to simulate a human entity moving around in, and interacting with, a PEIS environment. This simulator ended up being based on The Sims 3, and how this was done is described. The reader is expected to have some experience with programming.
Acknowledgements

I'd like to thank my supervisor Alessandro Saffiotti and my testers Federico Pecora Marcello Cirillo and Jonas Ullberg. Marcello deserves extra credit for the very helpful feedback he gave me on my documentation, and also for trying to get the Gazebo computer in the AASS lab to connect to the internet. Also thanks to Per Sporrong for miscellaneous help in the AASS lab. And last thanks to Tobias Nurme for keeping me company and letting me borrow his stuff.

Also thanks to the ModTheSims community for the helpful info they provided me with during my research on modding The Sims 3. And a special thanks to Kolipoki, lemm101 and BailaBaila99 for their very helpful tutorials.

Finally I'd like to thank myself. This project wouldn't have been possible without me.

Örebro the

____________________
Erik Westholm
5 Testing and Evaluation ................................................................. 21
  5.1 Approach to testing................................................................. 21
  5.2 Results of testing................................................................. 21
  5.3 Evaluation.............................................................................. 22

6 Conclusion .................................................................................. 23
  6.1 Limitations and Problems...................................................... 23
  6.2 Future Work............................................................................ 24

7 References .................................................................................... 25

Appendix: DOCUMENTATION FOR THE SIMS 3 SIMULATOR ........ 26
1 Introduction

This is the report for my Master Thesis project, which was performed in the spring of 2010. In this section of the report I will introduce my project by presenting the background information for the project, defining the problem and present our intended solution to this problem. I will also give a short description of the phases the project went through and the outline for the rest of the report.

1.1 Background

1.1.1 PEIS Home

Örebro University have created a room called the PEIS home. Apart from the fact that this home is really small and that it has no toilet it looks like any bachelor's home. It's got a bedroom, a kitchen and a living room with a TV. The interesting thing about this home is that it's been enhanced with PEIS ecology. PEIS is an abbreviation for “Physically Embedded Intelligent System” and it is defined as a set of interconnected software components residing in one physical entity [1]. These entities are by definition heterogeneous, but they are organized as one entity through the use of a common PEIS kernel. Through this kernel the devices can communicate with each other.

A PEIS ecology is an approach to realize an intelligent environment through the use of multiple PEISes. A simplified explanation is that it is a collection of PEIS entities that are connected to each other through the PEIS middleware. In other words, these entities can communicate with each other and you can also implement reasoning into the kernel and have the entities collaborate to solve advanced tasks. This collaboration between entities is one of the main fields on which Örebro University is doing research using the PEIS Home.

Many of the entities in the PEIS Home are sensors. For example, there is a sensor that keeps track on if the kitchen door is opened or closed. There is also a sensor that notices if someone is lying on the bed. All this sensor data can be used to try and figure out what is going on in the PEIS home at any given moment in time. This is the second main field on which Örebro University is doing research using the PEIS Home, namely activity recognition.

1.1.2 Activity Recognition

Activity recognition deals with automatically inferring human activities from sensor readings. We humans do this intuitively. If we walk into the kitchen and we see our room mate/family member at the sink chopping onion at the same time as a frying pan is being heated on the stove we can quite easily figure out that our roommate is cooking.
Implementing this capability as an automatic procedure requires the use of advanced Artificial Intelligent techniques. Interpreting the set of data can be quite hard, especially because of the nuances in the real world which will show in this set. Dirty readings, accidentally triggering sensors and taking a break in an activity to do some other small task are all things that complicates activity recognition. It's also difficult to create software that considers all possible activities available.

The goal of the research is to be able to implement assistive environments for elderly people. Activity recognition can be used to see how the elderly are doing (possibly from a remote location, which can be nice for relatives) and to figure out if they need some kind of extra help.

1.2 Problem

There is a quite big bottle neck in the research on activity recognition at the moment and that is the lack of decent data to experiment with. The problem lies both in quality and quantity. At the moment you can't find the right kind of data nor the right quantity of that data.

What can be done at the moment is to walk around in the PEIS Home and do normal things, like cooking and watching TV. But this is both time consuming and frustrating. You don't want to undergo 30 minutes of walking around in the PEIS Home every time you add a small tweak to your activity recognition program. That kind of data collection is for state of the art Activity Recognition, not for initial research.

1.3 Specification and Goals

The answer to this problem is to build a program that simulates a human being moving around a PEIS Home and doing normal things. This way data can be collected quickly and quite easily from the researchers own computers. If something goes wrong in the experiment it's easy to just restart the simulator, as opposed to the real world where you will have to physically set everything back to it's original positions. You will also be able to easily and effortlessly acquire a big amount of data.

Other advantages of having a simulator includes the ability to implement devices that don't exist in the real room, the ability to speed up or slow down time, the avoidance of unwelcome outside disturbances and the avoidance of mechanical errors\(^1\).

1.4 Planned Phases of the Project

The project took place in four phases. These phases was Background Study, Implementation, Testing and Documentation. Guessing what I did during these phases in general terms probably isn't very hard. Here are some details on what I did though.

\(^1\) See reference \([2]\).
During the Background Study I acquired a little knowledge on Activity Recognition (and the likes) so I had a better mental image of what was needed to perform the task at hand. I also researched possible software to do my simulator in this stage.

During the implementation I implemented my simulator. It involved, amongst other things, coding. It also involved programs I'd never used before and programming languages I'd never heard of before\(^2\).

During the testing phase I simply tested my software and the documentation of my simulator to see if it was understandable and usable.

During the documentation I wrote this report. But I also wrote a documentation of my simulator, and that documentation is almost as long as this report.

### 1.5 Outline of the Report

First I will go through the result of the Background Study by giving you info and evaluations of the four software alternatives I looked into. Then I will go through the result of the first step of the implementation, meaning the planning of the implementation. Then I will describe the actual implementation. Then I go through the testing and evaluation of the simulator I've created, and last I give you conclusions of the project (what was good, what wasn't and how can my program be made better).

\(^2\) Does Intermediate Language ring any bells? It didn't for me. You'll find a little more on IL in the third chapter of this report.
2 Possible Tools

When we (me and my supervisor) started the project we didn't know which software to use. My supervisor wasn't very familiar with the possible software at hand so it was my job to research the software and figure out which one was the most suiting software for the project. As mentioned in the specification I was to simulate a human being moving around in and interacting with a PEIS environment. So I was looking for software that could easily be made into such a simulator.

2.1 Gazebo

Gazebo is an open source robotics simulator. It's capable of simulating a population of robots, sensors and objects and do so in 3D. It comes with rigid-body simulation so that objects can interact with each other in a plausible manner [3].

2.1.1 Pros

- The PEIS Home is already implemented in Gazebo simulator.
- It's open source, so there are probably no limitations to what you can do short of limitations in programming languages at the moment. There's also decent documentation and an already functioning software to tweak, so most things we want to do can be done if given enough time.
- Proper physics simulation (more or less). This ensures that a realistic sequence of sensor triggers will be actuated. With luck we might even trigger a few faulty sensors “by mistake” (oops, I fell over the bed while moving to the kitchen!) to increase the realism of our simulator.
- Big potential. If the simulator is ever finished (and finishing it will probably take more than my project) it will probably be a quite impressive simulator which gives realistic results.

---

3 Section 1.3.
2.1.2 Cons

- The only sensors that are implemented at the moment are the cameras. This means that I will have to implement a few sensors, like a pressure sensor in the sofa and bed.
- Implementing a human is a non-trivial task. Only getting it to move around will prove a bit daunting, and to improve on this so that it interacts with the environment will prove even more daunting.
- Adding to the simulation will take time. For instance, adding another room is a project that will take weeks.
- The software seems quite demanding, especially if you want more than one camera. Though this might not be an issue when the computer being used doesn't use a six year old budget graphic card.

2.1.3 Evaluation of Gazebo

Gazebo is somewhat suited for our purpose. If given enough time the correct sensors can be added and once that is done we can add a human agent who has the capability to trigger these sensors. This will give (more or less) realistic sensor inputs for the activity recognition algorithms to work on. The time frame in which to do this job might not be the best but it is good enough and, most importantly of all, the time frame is reliable.

2.2 The Sims

The Sims is a game series developed by Maxis and published by Electronic Arts. It is a game where you control virtual humans, called sims. These sims live in a simplified simulation of real life, so they have a home, they need to eat and sleep and things like that. You work to make money and with this money you can buy better stuff that better fulfills your sims needs (a bigger TV is funnier to watch for instance). The version used in this project is The Sims 3.
2.2.1 Pros

- Human agents already implemented in the game. The sims interact with their environment, they make food, sit down on sofas, sleep in beds, watch TV etc. There's no need to implement a human agent if we go with The Sims.
- The human agents are also autonomous. Sims will get on with their lives pretty well on their own. As long as you've gotten them a job and have bought the necessary items to live a normal life you can leave them alone and they'll do alright. You can also control them yourself and get them to do what you want, which means our users won't be restricted by what their sims wants to do.
- It's easy to build houses. Since the game is what it is you can easily build a house your sim can move around in. If you want a bigger house you can expand it. This means that a big simulation environment can be built quite easily.
- Plenty of info on how to mod The Sims online. It's one of the most popular game series ever, and it focuses a lot on being creative. So there's loads of people creating their own content. And these people join forums, create tutorials etc so finding help won't be too hard.
- High usability. It's a game, so it's been designed to be easy and enjoyable to use.

2.2.2 Cons

- Strict limitations of what you can and can't do. Maxis have strategically opened up the possibilities to mod content for the Sims, but they have just as strategically shut some possibilities off. For instance, getting data out of the game is a bit complicated. The engine only uses things the Sims needs to run properly and not a single thing more. And examples of what it doesn't need is sockets and file streams. It's not impossible to get data out of the game, because error logging functionality exists but accessing this data requires a translator program of some kind. Getting data streamed in real time for our users to interpret is also a bit tricky. I'm not even sure you can access this error log while the Sims 3 is running.
- Uncertainties about what can and can't realistically be done. Most modders only create their own items which has it's own functionality. They know what they need to know to do this but not more. Altering the game on a deeper level is therefore something they don't know anything about. So if I, for instance, want to turn money off (would make things simpler for the users) or if I want to alter the time of the day then this is possible in theory (because you have access to all the gameplay code), but it might be too hard to find or even too hard to alter once I've found where to do it.
- Simplified behavior. When cooking the sims produces a cutting board with food from the fridge, works the cutting board with a knife for a few seconds and then puts the sauce pan the cutting board changed into on the stove. After only a few seconds the sim removes the sauce pan from the stove and it turns into a plate with food. This behavior isn't very realistic, and this might affect
the quality of the data this simulator would provide. Modding this is possible but it might be too hard.

2.2.3 Evaluation of The Sims

The Sims has big potential. If everything goes as planned then there will be a simulator up in just a few weeks that can do a lot and which is very easy to use. However, we might never get a simulator that gives us all the info we'd want or we might not be able to use this simulator in real time.

2.3 Second Life

Second Life is a 3D virtual world launched in 2003 by Linden Research. It's a free applications where users connect to a virtual world where they control an avatar. They can use these avatars to interact with the world and with other avatars. It aims at being a place where you can have adventures, see fantastic places and to interact with other people [4][5].

2.3.1 Pros

• Simulations of human agents are already implemented in the software.

2.3.2 Cons

• There is no real information available on how to mod Second Life. This means that if I was to go this route I'd spend one weeks just trying to figure out where to start. Because of this it's not worth looking further into the suitability of Second Life.
2.3.3 Evaluation of Second Life

Because of the lack of info on how to mod the game it's not worth examining further. For all I know the behavior in the game might be perfect for what we want, but due to the lack of info I still don't know if we can mod the game to utilize that behavior.

2.4 OpenSimulator

OpenSimulator is an open source server platform for hosting virtual worlds. It's most famous feature is it's compatibility with Second Life, but you can use it completely separate from Second Life if you want to as well [6][7].

2.4.1 Pros

- Simulations of human agents are already implemented in the software. At least to an extent.
- It's open source.

2.4.2 Cons

- It's still only in Alpha Stage. This may mean that the program is unstable.
- Poor documentation. Just like with Second Life it's not easy to find out how to develop for OpenSimulator.
- Simplified behavior. Even though you do have a human who moves around it's not very complicated. Sitting down isn't done by the act of sitting down, it's done by one instance standing at a point and the next you sit at the place you selected for sitting down. This might be tweakable though, but that's something that has to be done if we choose to go with OpenSimulator.
2.4.3 Evaluation of Open Simulator

Just like with Second Life there wasn't enough info available for a proper evaluation.

2.5 Conclusion of Evaluation of Possible Tools

Pretty early it was clear that we were choosing between Gazebo and The Sims. This was because neither OpenSimulator nor Second Life had enough info available to be able to properly evaluate them.

As for the Sims and for Gazebo The Sims got the upper hand, since it appeared that implementing a simulator in the Sims seemed easier than to do it in Gazebo due to all the things you'd get for free. It was also a much more interesting alternative for the simple reason that it was a game.

There were things that concerned us though, mostly because of limitations in what Maxis allowed to do with mods. But in the end we decided that it was still the best option, because it appeared that the worst case scenario (no proper position estimation and no real time) was still something that could be useful for activity recognition research.
3 Planning the Implementation

3.1 Details on The Sims 3

The Sims 3 is (at the time of this project) the latest installment in the Sims series, a series developed by Maxis and published by Electronic Arts. According to Wikipedia it was released on June 2, 2009 in North America, on June 4, 2009 in Australia and on June 5, 2009 in Europe. Just like the previous two installments you control Sims who have homes, careers, interests and relationships just like real life people. The big advancement The Sims 3 have made over the older games is the big, open world that the sims now live in. The sim live in a small town with everything a real small town has (like restaurants, a supermarket etc), and the sim can leave his home lot to go to sporting events or the library or maybe just to the central park to hang out. The entire town is run in real time when you're playing, not just your own controlled sims.

That is not the reason I chose to mod The Sims 3 over the other two versions though. The reason was that my research suggested that not only has the game itself been improved, the internal structures of the game and the modding possibilities of the game have both been streamlined to make it easier to make your own custom content.

3.1.1 What makes the game modifiable

All of the game content has been placed in package files, which is a kind of database file that contains info on what files should be used to run the game. This is a file format used by many of Maxis games, including SimCity 4 and Spore [8]. With the right software you can extract the DLL's where all the gameplay is implemented from these package files, edit these DLL's and put the altered dll's back into the game. You can also make your own package files, containing references to your own DLL's, that enhances the game.

3.1.2 Execution Plan/Requirements

One of the first things I had to figure out was how to go about to implement a Human Behavior Simulator in The Sims 3. Here is where I describe the plan I came up with.

The base plan was the basic plan I set up for building my simulator. Once the plan had been implemented the Simulator would have the basic functionality required to collect data for Activity Recognition. The Optional steps were merely bonus steps that would enhance the Simulator.

Note that the plan is based on the idea that there is an error log which you can write to. In the core files this isn't the case, but I managed to find an already existing mod that did this for me. What this mod did was simply to reintroduce some debugging functionality that had been removed for the final release of the game.
Base plan:

1) Mod one item so that you notice when it's being used (or, when it starts and when it stops). For instance, mod a stove so that it makes a sound when it's turned on and when it's turned off.

2) Mod the same item so that it writes an entry in the error log instead of playing a sound when it starts and when it stops being used.

3) Mod items like a TV, a stove, a bed, a shower and a toilet so they all write to the log when they start and stop being used. Also look into modding kitchen tables, sofas and chairs.

4) Add position estimation functionality. Possible solutions here are writing to the log whenever a sim position is updated, modding doors so that they trigger that they've been opened and adding burglar alarms that writes when someone enters and leaves the room.

5) Implement a middle program that keeps track of the current state of the world by reading from the log. If we can't run The Sims 3 in real time this interpreter could simulate the game flow by sleeping between packages.

The middle program wasn't really a requirement. The users could have written the code for handling the information that comes out of the game themselves, but it made sense to me that I wrote the program that did this and that the users merely used the information I collected.

Optional steps (note that these steps were written before I had a clear view of what was and what wasn't possible to change in the game):

1) Mod even more items. A computer is a good idea, for instance.

2) Possibly mod a carpet so that it notes when someone walks over it. This could be a way to handle position estimation functionality.

3) Add control over what time of the day it is.

4) Add control over the mood of the sim. This would enable the user to (for instance) make the sim really tired when the user wants the sim to sleep, hungry when the user wants the sim to eat and so on.

5) Turn off money/make everything free. Just so we don't have to write the cheat code for getting extra money all the time.

6) Simplify the game. There are numbers of unnecessary things that we don't need in our simulator. Examples include turning off fire in stoves and turning off quick meal. Others include making the loading of your game easier, the possibility to turn off visitors/outside activities, turning off aging, turning off the update of physical updates of the body (you can get fat if you eat too much in The Sims 3).

7) Implement ideas from the users. I'm sure they feel something should be in the game and then I should try to implement it.
3.2 Software

Here's is where I describe the software I used in this project and what its use in the project was.

3.2.1 S3PE

S3PE stands for The Sims 3 Package Editor. It is a program that reads, edits and creates package files. When you open a package file you see all the references (in lack of a better word) contained in the package file and you can add, edit and remove these references. You can also export a reference, which means you copy the file the reference is referring to to wherever you want to put it.

According to simpedia it was made by Inge and Peter Jones [9].

3.2.2 S3OC

S3OC stands for The Sims 3 Object Cloner. It's used for taking object in the game (like a fridge or a sofa) and clone it to a Package file which I can alter without changing anything in the original game. By writing my own DLL in C# I can also add advanced behavior to my cloned item.

This program was, according to simpedia again, made by Inge and Peter Jones [10].

3.2.3 ILASM and ILDASM

These two programs are part of the Microsoft .NET framework and they are used to assemble and disassemble .NET assemblies. In this project they are used to disassemble the game DLL's to editable IL files. IL in this case stands for (common) Intermediate Language, and it is the standard .NET intermediate language. An intermediate language is the language C#, C++ and C (and others, I'm sure) are translated to when they're being compiled in order to optimize the code. It consists of a lot of push, pop save to, load from and jump to commands.

You can read this code but it's not all that easy to do. You can also edit it, but this also isn't all that easy to do. Despite that, extracting a game DLL with s3pe, disassembling it, editing the code in IL, reassembling it and passing the new DLL back into the game with s3pe is a possible (abet complicated) way to change the game's core functionality.

3.2.4 .Net Reflector

.NET Reflector is a quite handy program that gives you the opportunity to look at a .NET component, such as a DLL-file, and look at the code that was used to make that DLL-file.
This means that we can take the DLL's we've gotten with S3PE, open them in .NET Reflector and see all the classes and code that makes up The Sims 3. This is extremely useful, since we can see what classes exists in the game, what methods they have and we can even see the code for these methods (written in C# or other language of choice). When you use ILDASM to disassemble the DLL-file into an IL-file this program is very helpful, since you can use it to figure out what the IL-code does. You don't need much knowledge of IL to figure out which part of the IL-code does what. When you've made your changes and assembled back the IL-file to a DLL-file you can open the DLL in .Net Reflector again, look where you made your changes and that way verify that you made no errors in your IL-coding.

Note that you can't see all the code that makes up The Sims 3 for safety reasons. For instance, the base file that runs the game with help of the package files is unreadable in .NET Reflector. And there are also parts of the rest of the base code that is hidden. Still, with this program you can learn a lot about how The Sims 3 works, and that way you can figure out which changes to make it work the way you want it to [11][12].

3.2.5 Notepad++

Notepad++ is a freeware text editor that works much like normal Notepad. However, like C++ is an improved version of C4 of improvements Notepad++ is an improved version of Notepad [13].

In this project it's only used to edit IL-files. It is by no means a requirement to do my job (IE I wouldn't have failed without it), any program that can read IL-files will do (including Notepad) but this one works and it's reasonably fast (Notepad doesn't).

3.2.6 Microsoft Visual C#

Visual C# is the C# version of Microsoft Visual Studio. It is a multi-language software development environment developed by Microsoft [14]. This was the tool I used to write and compile the code for the DLL-files used by my cloned objects. The free version of Visual C# gave me all the usability I needed to preform the task at hand.

3.2.7 Eclipse

Eclipse is, just like Visual Studio, a multi-language software development environment. It's primary used for Java programming but, with the right plug-in, you can use it to develop in pretty much any language out there. I used it mainly for Java Programming [15].

---

If we simplify how C++ relates to C.
4 Implementation

If you look at the plan presented in section 3.1.2 you'll see that the first five steps involves changing the behavior of The Sims 3. The reason these were the first five steps was that I simply didn't know if it was possible to do what I wanted with The Sims 3. In other words I went for the biggest uncertainty first so I'd find out early if it didn't work.

There were a few key steps to modding The Sims 3. Getting data out of the game was the first and the most important one, since the project would be impossible without it. The game's standard libraries didn't allow for this to happen (as I mentioned in chapter 2 the game only allows what is necessary for the game to run, and apparently writing to files isn't). Therefore I needed to find some outer source to get that to work.

The second step was to mod items so that they sent signals when they were being used. This would be done for enough items to be able to build a normal home with modded items. The third step was to add some kind of position estimation to the simulator. The fourth, and last, step was to build a program that was easy for the future users to connect to and that would provide the users with the state of the game whenever the state changed.

After these four steps was preformed I needed to write a documentation of the simulator. I could also add more functionality to the simulator if I had time.

4.1 Getting data out of the game

The first step was to figure out if it was possible to get data out of the game. I'd asked around online during the research part and gotten hints that a mod existed that reintroduced some debugging functionality to the game that, among other things, let you write things to a file. And looking around online I found that mod, called EnableScriptError.

With this mod in use you could call a function named EnableScriptError. To see what this function did, let's take a hypothetical sofa and say that we modded this sofa so that every time someone sat down EnableScriptError would be called (how is not important at this stage). What then happened in the game when someone sat down in the sofa was that a big, blue error window would be displayed. It gave you info on what had happened, where it happened and it also gave you options on how to handle the error. The options was things like “write to file”, “ignore” and “delete object”. Pretty much whatever you chose the behavior itself would stop, meaning the sim wouldn't sit down in the sofa.

This wasn't really the behavior I was looking for. I was looking for the function to simply write an error of choice to my error log and then do nothing else. This meant that I had to change the function. And in The Sims 3 the only option when it comes to changing an already existing mod (especially a mod based in the core Sims 3 files) is changing the DLL itself. Which meant disassembling it with ILDASM, making the modifications in the resulting IL-file and then assembling it back with ILASM. This is
the only way to go because the game won't use multiple versions of the same file (for quite obvious reasons). So, you either rewrite the entire thing from scratch in C# or mod the core file. Which means you mod the core file unless you have two weeks of spare time with nothing better to do (the short time is due to the fact that you can copy things from .NET reflector).

As for the actual rewriting of the function, step number 1 was getting rid of the script error window. This was a matter of going to the IL-code and cutting out everything but a call to the log function and a return true call (I figured this out in Reflector). And when I'd done that and the sim sat down the only thing that happened was that a script error was written to a file. There was no hint in the game that this had just happened, the sim behaved normally.

However, the log function didn't work quite as I wanted it to either. The first problem was that every script error was written to a new file rather than all script errors being written to the same file. The second problem was that most of the info written to that file was (from my point of view) totally useless. This meant more rewriting of functions.

I started with changing what info was written to the log. This function took an exception, and then it wrote that exception plus a bunch of uninteresting info to a file. Once again, I basically removed everything apart from the code that wrote the exception to the file. Next up was changing where the error was written to a file and simply change the file path. Here I ran into problems, however. You see, I couldn't find the code where the error was written to a file. I spent two days looking for this code but it just wasn't there. So I figured this code had been placed in one of the closed DLL's for safety reasons. This meant that I had to make do with having the errors written to one file at a time.

After my modifications to the error logging functions I was left with the ability to call a function called EnableScriptError, which took an exception as argument. When this function was run a script error file was created that contained that exception. This meant that if these files weren't handled somehow there would be a big pile of script error files inside the Sims 3 folder in My Documents (that's where the files were created). This spoke further in favor of having a middle program as an interface for users to interact with. This middle program could take care of handling these script error files so the user didn't have to.

The format for all my errors was Time*Item ID*Status. Time was given in the format date followed by hour:minute:second. This is the internal format available in the game. The item ID consists of a long string of arbitrary letters and numbers. This is also the format available in the game. The status is either on or off.
4.2 Item Modding in The Sims 3

4.2.1 Initial approach to item modification

My initial idea on how to introduce sensors into items in The Sims 3 was what's probably the most intuitive and simple way: I tried to use s3oc (the object cloner) to clone items and then add calls to the script error logging inside the C# code I made for my cloned objects.

An example is my first modded item: a sofa. Inside of the sofa class there are two functions that I'll call SitDown and StandUp for simplicity's sake (their names are longer in the real code). In my modded sofa I modified SitDown and StandUp so they're basically a call to EnableScriptError followed by base.sitDown and base.StandUp. This worked just fine for sofas. Note that I had to change the core files to make SitDown and StandUp overridable.

When I tried to do this for a bed I ran into problems though. This was because, due to the existence of SitDown and StandUp for sofas, I missed the internal structure for items in The Sims 3.

Items in The Sims 3 can be used by sims, and this is done through interactions. For instance, when you're playing The Sims 3 and you click on an item you get a menu with things you can do with that item. For sofas this menu generally displays two items: sit and nap. When you click any of these the sim will walk up to the sofa and use it the way you told the him to. IE if you click Sit the sim will walk up to the sofa and sit down.

The code for these interactions is placed inside of Interactions, which is an interface. For example, Nap is a class that implements Interaction. In the sofa, all you see of the code that implements Nap is that a pointer to the Nap class is added in the Sofa initiator. In other words, all the code that's used for running the interactions in the game is in fact run in sub-classes that doesn't really have anything to do with the item itself.

So, when I tried to modify the bed I realized that there was no clear idea of where I could add my error log calls. There was no LieDown or StandUp functions. All that was handled inside of the interactions associated with Bed. This meant that cloning a bed wasn't an option, because doing so meant that I had to copy all the interactions into my DLL as well. This was possible, but it would require a lot of copied code with high risk of breaking something on the way.

Instead I decided to try and modify the core file instead. I had the IL-code for how the call to EnableScriptError looks thanks to my modded sofa, so I pretty much took that code and copied it into the interactions of the bed. With a little tinkering this worked too. And it didn't only work, it worked for all beds in the entire game!
4.2.2 Re-evaluation of approach to item modification

When I'd gotten to modding the bed I already had two cloned items working, a sofa and a fridge. And I noticed that every time I made a change in the core game I had to rebuild the DLL's for these two items and then import the newly built DLL into the corresponding package file. This made me realize that as my modding progressed and I cloned more and more items, the longer a change in a core file would take to update since I'd have to rebuild more and more items every time this happened.

Along with my realization that modding the behavior inside the core files wasn't all that hard made me re-evaluate my approach to modding the game. Instead cloning items and modding those clones I went into the core files and added the new code there. This had the advantage that updating a change was a lot easier and the modding also covered all items of a certain kind rather than just one item. Modding the behavior of 'bed' meant that all beds got modified, not just one kind of bed.

Cloned items was restricted to items that required some kind of extraordinary behavior that wasn't easy to preform in IL.

4.2.3 Modded items/behavior

Simple error logging functionality has been installed into sofas, beds, toilets, showers, fridges and stoves. How this would appear in real life varies from item to item. The sensor in the sofa would be a pressure sensor that triggers if someone, for instance, sits down in it. In the fridge and shower this would instead be a sensor in the door that notes when the door is opened. It is so far a quite modest collection of items but adding more items to the list wouldn't be at all hard.

Adding position estimation was a bit tricky. The sims do have a position value, but to keep track of this position appears to require tinkering in the core mods, which is something that's best avoided as much as possible. Instead I went for a simplification of the problem: rather than keeping track of exactly where the sims are I just keep track of which room they are in. This could be achieved by cloning a thief alarm and changing it's behavior so it wrote to the log each time someone (no matter who) entered or left the room. Since I could write the code in C# this was fairly easy to accomplish.

4.3 The Master Program

The Master Program is the name I gave to the middle program responsible for assembling and interpreting the Error Log files written by The Sims 3. This program existed in my mind at an early stage, since I thought it would give me the ability to create an easy interface for the users to interact with. But as soon as it was clear that the only way to get info out of the game was to write a lot of error log files (one per message) this program also became a necessity.

The program was written in Java. The reason I chose Java was that it was one of three main languages used in the AASS lab and of the three main languages this was the
one with wide multi-platform support. You can, if you want to, connect to my Master Program from a different computer on a different OS than Windows if you use Java.

The game state is represented in its own class. This class consists of a list of items (which all have an ID, a name and a value) and supporting functions. In this Game State the programmer will hard code which items in the game corresponds to which item in the Game State. For instance, which ID corresponds to the bed, which ID corresponds to the fridge etc.

4.3.1 Underlying structure

The main idea of the Master Program is that users connect to it and the program supplies the connected users with updates when the state of the game changes. The connection is done through an ordinary server which is always running.

The program will continuously try and update it's Game State, and when an update is made it sends the update to all connected clients. How the updates are preformed depends on which mode the program is in. There are two modes.

The first mode is The Sims 3 mode. In this mode the Master Program will simply look in the folder Documents/Electronic Arts/The Sims 3 to see if there are any ScriptError files there. If there are files there the program will read each of these files, write the message in the file to a separate log, delete the file and process the message. Processing the message means checking if the ID in the message matches any item in the Game State. If it does Game State checks if the message value is different from that item's current value. If it isn't the state has changed and the new value is sent to all the clients.

The reason it checks if the ID matches any item in the game state is that every item of a modded class (stoves, sofas etc) emits script errors. So, there are a lot of script errors that will be written which the Master Program should ignore.

The second mode is Script Scheme mode. In this mode the program won't look in the folder Documents/Electronic Arts/The Sims 3 for ScriptError files. Instead the program will run Script Scheme files, which is basically files that contains multiple Script Error messages (the log can quite easily be used as a Script Scheme file). When a Script Scheme file is run the program reads the first message of the file and processes it, reads the second message, waits based on the time difference between the two messages, processes the second message, reads the third message, waits based on the time difference, and so on until all messages has been read. This mode makes it possible to collect data in one place and run the data on another.

A user can for instance go to a Windows computer with the simulator installed and collect some data using The Sims 3. He can then take the log file to his Linux computer and test his Activity Recognition program on that data. If there's a bug in his program he can correct that bug and then try it out again on the same data without having to redo the data collecting in The Sims 3.
A user can also, due to how I've implemented the GUI, set up a long sequence of files and test on that sequence. Since I've collected data of mundane activities like “Eat lunch”, “sleep” and “go to the bathroom” the user will be able to simulate long sequences of data without having to set it all up manually inside The Sims 3. These sequences can be randomized in large quantities. You can also increase the speed, making testing quicker.

The program keeps track of two separate game states. one for The Sims 3 mode and one for Script Scheme mode. When the server switches mode the correct scheme will be sent to the users.

4.3.2 GUI

The GUI is a very simple thing. It can be separated into two sections, where the first section is the part where you switch between The Sims 3 mode and Script Scheme mode. It's quite obvious how this works. As long as you're in The Sims 3 mode the program will read files from Documents/Electronic Arts/The Sims 3, and the rest of the GUI is turned off. When you go to Script Scheme mode the program will stop reading from The Sims 3 folder and the rest of the GUI will activate.

The rest of the GUI is where you control the Script Scheme mode. The idea is that when you hit the Run Scheme button the files that have been written into the text field will be run, one row at a time. If a row contains a file name that can't be found inside the set folder the file won't run.

You can create a scheme by either writing the files you want to run in the order you want to run them or you can make a random scheme by checking the files you want in
the file list (checking no files means all files in the folder will have a chance to be included in the scheme) and click Create Scheme. By setting the value right of the files you can make it more likely that a certain scheme is run. Once you've created this random scheme you can modify it manually if you want to.

4.3.3 The Client Shell

The Client Shell is a standard client that connects to the server and then waits for messages to be sent. When it connects to the server the current state will be sent from the server. The client will then create a GUI that consists of two labels per item in the sent state, one for the name and one for the value. The value gets updated whenever an update is sent from the server.

The Client Shell initially does nothing but display the current state. The idea is that the user shall be able to combine his own Activity Recognition software with this Client Shell.
5 Testing and Evaluation

Once I'd built my simulator it was time to start thinking of what would happen to it once I was done with it. People will want to use it and they will also try to improve it further\(^5\). This meant that I had to document how to install, use and develop my program.

This documentation was quite big, and to do most things described in it requires several unintuitive steps. For instance, installing requires that you not only make a standard installation of a game, you also have to run two separate files, copy two different folders to two different places and (especially) that you rebuild three IL-files and import them into my mods in s3pe. Which from scratch means you have to install s3pe, find the right ILASM and copy those to a folder, copy the IL-files to the same folder, assemble them using the command prompt, and then go into s3pe and import the newly assembled DLL's to the correct packages. Sounds complicated? It is.

So, in order to make sure the users understood how to install, use and develop my simulator I had to test my documentation. There was simply no other way to verify whether my simulator would be usable for other people or not.

5.1 Approach to testing

The approach was a pretty straight forward “dump the files on a user and see if he manages to figure out what he's supposed to do”. The user would then give feedback on what he did/didn't understand and what was/wasn't clear. Due to the fact that two of the three testers are university teachers this approach worked very well since they'll be able to spot what's good and what's bad instantly.

5.2 Results of testing

The result of the testing was positive in the sense that my users managed to figure out what they were supposed to do. The first test person had quite a lot of comments on my documentation, but after I'd corrected those the second tester only had some minor comments. At this point I concluded that my users would at least be able to install and run the simulator without problems.

Modding the game is a different matter. None of the testers really tried that part out so I only corrected that section based on the hints the first tester had given me. I hope that the instructions are clear enough to help, but the worst case scenario is that they'll have to figure that part out themselves like I had to. And I think my instructions are good enough that it's far from that bad.

\(^5\) In theory, anyway...
5.3 Evaluation

If we look at my initial plan (check 3.1.2 to see the plan) and compare it to what I actually accomplished we will find that that the first five steps have been implemented. When my mods are installed The Sims 3 will write to a log file when someone uses a fridge, a stove, a toilet, a shower, a bed or a sofa. And my Master Program collects these log files and through those keeps track of the current state of the game. Clients can connect to this Master Program and get updates whenever the state of the game changes. Through a modded burglar's alarm we can keep track of how many sims are in each room of our simulator.

A GUI has been implemented, and in that GUI you can play Script Scheme files. You can write a list of many files that the program is to use, and you can randomize this list. This was made after users commented that this kind of feature would make testing of their programs a lot easier.

I also wrote a big documentation that explains how to install, run and improve my simulator.

The users found the GUI to be reasonably good, if not great. It filled it's purpose, but a few helpful features was mentioned as possible improvements. For instance, being able to change the path to where the Script Files are written in the GUI and also modifying the list of items in the Sims home from the GUI. At the moment you have to modify these in the base code, which is a very lengthy and cumbersome way to do it. These two suggestions made me conclude that the first thing to do in order to improve my simulator would be to add some kind of settings functionality that stored these things on a file. Obviously this would be accompanied by some kind of settings window in the GUI that can modify this file.

The judgment of the simulator itself was that it would be very good for debugging purposes. It appears to be unsuitable for realistic experiments though. This because sims do not perform complex tasks, and the tasks are always carried out in the same way.

From this you might conclude that I've done a poor job. But, given my prerequisites, this was as much as was expected of me. A Gazebo simulator would have resulted in more or less the same kind of simulator. IE a simulator that wouldn't have been suited for anything but debugging. The difference would have been that Gazebo could have been enhanced into something more. Doing that with The Sims 3 will be difficult, if not impossible.
6 Conclusion

At the start of the project it was asked of me to make a simulator of a PEIS environment with a human being walking around in said environment triggering sensors. The resulting data was to be used for research on activity recognition.

At the end of the project I've managed to accomplish that. I have a simulator in which human beings walk around and use objects, and when these objects are used sensors are triggered. I've also written a program that keeps track of when these sensors are triggered, and by connecting to this program the users can acquire this information. I have also manged to add functionality beyond the initial requirements that users claimed would be useful.

6.1 Limitations and Problems

The biggest problem involves having more than one person use an object. This is due to the fact that we're not checking the correct thing. The correct thing would be to have some kind of sensor that continuously checks if an item is in use. This is how our modified burglar alarm works. Every iteration of the game the burglar alarm checks how many sims are in the room and if the number has changed since the last iteration the burglar alarm writes to the log.

The sofa instead signals when someone sits down and someone rises. Apart from being an inaccurate representation of reality it also leads to problems if two sims use the same item. For instance, if two sims sit down in a sofa and one of them rises the sofa will be set to 'off' even though there's still one person sitting in the sofa.

Another problem is with items that mainly has their error logging code inside interactions. If you're resting on a bed, then decide to take a nap on the same bed, the sim will go from “resting” to “napping” without getting up from the bed. But signals will still be sent when resting stops and napping start, so it will look like the sim really quickly rose from the bed and then laid back down.

A clear limitation is that you don't get the exact position of each sim, you only know how many sims are in each room.

Another limitation is that the GUI isn't exactly a bastion of usability and visual quality. This doesn't make the program unusable (not close, I'd say), but that you use text input to write the script schemes doesn't strike me as the ideal solution. I think some kind of drag and drop solution would be much better. There are other things that could improve the GUI as well.
6.2 Future Work

This is the simple list of things I'd suggest doing to improve the simulator:

1) Improve the GUI. This should be done by adding some kind of settings file that's loaded with the program and that can be manipulated within the GUI. The two main things that should be added is the path to where The Sims 3 puts the script error files and a list of items to keep track of.

2) Another thing should be added to the GUI: A better visualization of the progress of script files. At the moment a script file says “success” when they're done, but while it's running you get no indication at all.

3) More functionality that could be added to the GUI is the ability to “record” script files while The Sims 3 is running.

4) And last, making the GUI look nicer. One complaint on the GUI was that most of the GUI deals with Script Mode, meaning they're useless in Sims mode. These should be placed in a separate tab, one that's hidden while in The Sims Mode.

5) Fix the problem of multiple sims in the apartment. Can be easily done by counting the number of “on” and “off” messages read in the Master Program.

6) Create an item that can set the sim's mood to what the user wants it to be. This can be useful for doing more free-form testing where you want to “make” the sim behave (or not behave) in a certain behavior.

7) Mod more items. The current list of items is a bit scarce.

If you're feeling really daring you could try and move all activity recognition to our modded burglar alarms. You see, what my alarm does is that it goes through the list of sims in the room it's in and simply checks how many these are, and if there are a different amount than there was a second ago it writes to the log. There should be (I'm not really sure, because this section is just a theory of mine that might work) a list of items just like there is a list of sims. And if these items have some kind of IsInUse function we could keep track of these items through the alarms. Simply create a list of all items in a room and check each item in the room against this list. If the item is in use and wasn't before (or the other way around) write to the log. This would give us proper behavior of the items as well.

Another theory I have is that it's possible to get the real coordinates of these sims through our modded burglar alarms. Instead of keeping track of how many sims are in the room we keep track of each sim in the room's position. If the position changes we write to the log. We might also be able to get the sim's posture this way.

I'm not sure if these last two are possible or not, but if they are possible they'd increase the realism (and accuracy) of my simulator.
7 References


APPENDIX: DOCUMENTATION FOR THE SIMS 3 SIMULATOR

By Erik Westholm
Index

1 Introduction 1

2 Guide to Installing the Sims 3 Simulator 1
   2.1 Installing the Master Program .............................................................. 2

3 Guide to Running a Client with The Sims 3 Installed 3
   3.1 Collecting data from The Sims 3 .......................................................... 3
   3.2 Running Script Files ............................................................................. 3

4 Guide to modding The Sims 3 4
   4.1 Modding items to write to the log ......................................................... 4
      4.1.1 Script Error Files ........................................................................... 5
      4.1.2 Editing the start/stop function ....................................................... 5
      4.1.3 Editing the interactions ................................................................. 8
   4.2 Custom Made Items ............................................................................. 9
      4.2.1 Rebuilding your Custom Items ....................................................... 10
   4.3 Remodeling the Sims 3 Home and adapting the server to the new home ......................................................................................... 10
      4.3.1 Updating the simulated home ....................................................... 10
      4.3.2 Updating the Master Program ....................................................... 11
      4.3.3 Adding script files to the Master Program ...................................... 11
   4.4 Hints and Tips ..................................................................................... 12
   4.5 Change log of functions ..................................................................... 12
   4.6 List of Modded Items .......................................................................... 13

Appendix A: Core Modding! (For complete and total nincompoops, like me!) 14

Appendix B: IL and DLL overview / tutorial 21

Appendix C: Object Modding (aka adding interactions) 36
1 Introduction

The Sims 3 Simulator is a simulator of a human inside of a PEIS environment, and was built to be used for data collection for research on activity recognition. It's based on The Sims 3, and the main role for the game is to do the actual data collecting. The relevant data in The Sims 3 gets to the user through error log files that are collected by a Java program. This Java program keeps track of the state in the game through these error log files, and it also works as a server which the user can connect to with their activity recognition software and get the info on what is happening in the simulated PEIS environment. The Java program also contains functionality for simulating a PEIS environment from files rather than from The Sims 3, which can be useful when you want to try your Activity Recognition algorithm on a lot of data.

This documentation contains info on how to install the simulator, how to use the installed simulator and how to modify the simulator.

2 Guide to Installing the Sims 3 Simulator

Installing the simulator comes in two steps: installing The Sims 3 and installing the Java Master Program. This can be done in any order, but I'll start with The Sims 3.

Disclaimer: If the game wants to update automatically, don't let it. Our mods are only compatible with version 1.3 of The Sims 3 and if you auto update it will update to version 1.10+.

Here's the step by step guide on installing The Sims 3 with mods:

1) Install The Sims 3. If you're not running Windows, you can check online if it's possible to install The Sims 3 under your OS. Please keep in mind that the folder structure is important. Also keep in mind that while The Sims 3 might be runnable on your computer modding the game requires Windows. All my separate files have been built using Windows and while a different set of modding software might exist for your OS converting from my files to the files you need might not be possible.

2) Update The Sims 3 to version 1.3.24. Do this by running The Sims 3 Patch 1.3.24.exe, which you can find this in InstallFiles folder in the same folder you found this document.

3) Install the Modding Framework. This is done by running FrameworkInstaller.exe in InstallFiles/ModFrameworkInstaller. When the program has started, check that The Sims 3 is pointing at the right directory (change to the correct directory if it isn't) and click install.

4) Install the mods. Do this by simply copying the folder Packages from InstallFiles to The Sims 3\Mods\Packages.

5) Copy the Save folder from The Sims 3 Installation Files to Documents\Electronic Arts\The Sims 3. If there is no Electronic Arts\The Sims 3 folder inside of Documents then simply run the The Sims Game Launcher and it will appear. As a side note, when you click the Game Launcher, if a popup tells you there's a new verision of the game out then click cancel and go to Game Updates and untick the Automatic Updates checkbox.

6) Update the package files. This step requires that you're using Windows and that you've got .NET 2.0 installed (you can download it from microsoft.com if you don't). If you're not using Windows you can do this step on a Windows machine and copy the newly made files to your current machine. First, find the files ilasm.exe, ilasm.exe.config and fusion.dll in the folder Windows\Microsoft.NET\Framework\v2.0.50727 and copy them into the folder IL
Files which you'll find in the Development map found in the same folder as you found this document. Next, start the command prompt and direct yourself to the IL folder using the cd command. Type the following three commands:

ilasm SimIFace.il /dll
ilasm Sims3GameplayObjects.il /dll
ilasm Sims3GameplaySystems.il /dll

You can copy these and paste them into the console by right-clicking and selecting paste.

Next, cut the DLL files that you just generated (SimIFace.dll, Sims3GameplayObjects.dll and Sims3GameplaySystems.dll) and paste them into a new map called DLLs, which you create within the IL folder.

Next we need to install a program called s3pe. Do this by running s3pe_0912-13-1729.exe which can be found in Development/DevelopmentSoftwareInstallationFiles. Next, open s3pe, click File/Open and go to The Sims 3/Modding/packages. Open enablescripterror_1.3.24.package (note that s3pe needs to run as an administrator to open packages). Inside this package file there will be one entry. Select this entry and click the grid button in the bottom of the window. Click the arrow in the Import/Export field and click Import. Direct to your newly created DLLs folder and double click SimIFace.dll. Next, press the commit button and save the package. Save the package. Repeat this process with the other two dll's you've created.

Next, go to The Sims 3\Game\Bin and open the file gameplay.package in s3pe. Make sure the preview radio button is set on value. Go through each field in the list and look in the gray window to the right until you find the one which Manifest Module field was UI.dll. Copy the manifest module name (in this case UI.dll), click grid, click the Import/Export arrow and select Export. Browse to the DLLs folder (Development\ILFiles\DLLs), paste the previously copied name in the file name field and click save. Open up Scripts.package and repeat this process for Automation.dll, ScriptCore.dll, Sims3MetaData.dll and TestObjects.dll. Then open SimCore.package and repeat the process for System.dll and System.XML.dll.

Next, go into Development/VSProjects/AlarmProject and open AlarmProject.sln in Visual C#. Go to Project→Properties→Build and change the output folder to Development\ILFiles\Custom Items DLLs. Next, click the plus next to references and select all the references. Press Delete and click OK to remove all the references. Next, right-click references and select Add Reference. Go to Browse and browse to the DLLs folder. Select all the DLLs in this folder and press OK. Watch as the reference list fills up. Next, press F6 to build the project.

Last, go to The Sims 3/Mods/Packages and open the file Ubereil_BurglarAlarm.package in s3pe. Find the file with the S3SA tag. If you can't see the tag field, find the type field. Move your mouse to the left edge of this tag, click and drag to the left. Now you should see the Tag field. Once you've found the S3SA file, press grid, arrow, import and go to the build folder of the AlarmProject folder. Double click AlarmProject.dll, press commit and save your package.

Now you can run The Sims 3 and it will, when certain items are used (see List of Modded Items at the end of the document), create ScriptError files in the folder Documents\Electronic Arts\The Sims 3. To find out more of these ScriptError files, go to the Guide to modding The Sims 3 part of this documentation.

2.1 Installing the Master Program

To interpret these ScriptError files we need to install the Master Program. This is done with ANT. So, make sure you've got ant installed, open the console, go to
Development\MasterProgramAndClientShell and type ant. This should compile the two programs and add the executable files to the dist folder.

3 Guide to Running a Client with The Sims 3 Installed

There are two modes in my simulator. In one mode the program tries to read files written by the Sims and that way keeps track of the state of the game. In the other mode the program executes script files from a folder. These two modes operate a little differently. But for either mode to work you start the Master Program and then you start your client which connects to the Master Program. Which mode you are in is showed by the label in the top left corner, and you switch between the modes with the big button next to that label.

Note that running a script file does not influence The Sims 3. The two modes are run separately.

3.1 Collecting data from The Sims 3

Once you've started the Master Program and connected your client, start The Sims 3. Do not update the game! When you arrive at the save selector the active save should be one called Sunset Valley with the family Rat. If that's the case just click play (round, teal button with a play symbol). If not, close the game, repeat step five in the install instructions, and repeat this step. Data will be collected from the Sims 3 as long as the label in the top left corner is set to 'Sims Mode'. In this mode any files read from the sims will be written in the text area to the left.

3.2 Running Script Files

In order to run Script Files you have to be in the Script File mode. If the label in the top left corner doesn't say Script File Mode, click the Go to Script File Mode button.

If you want to run a script scheme you will first have to create a scheme. This is done by writing the list of files in the text field in the left edge of the window. You write one file per row, and you have to write the full file name. Once you have your list you click Run Scheme and the scheme will be run. You can increase the speed of this run by increasing the number in the Speed Combo Box.

You can auto-generate a scheme with the make script scheme button in the down right corner of the window. The number of entries that will be generated is equal to the number in Number Of Files To Be Run Combo Box. If you check two or more of the files in the File List only the checked files will be used when creating the scheme. If you want one of the files to occur more frequently than other files you can increase the priority of that file by increasing the number in the combo box right of it's name. Note that this generation will ensure that the same file isn't written twice in a row.

If you add more Script Scheme files to the folder as you run you can click the refresh file list and these files will be added to the file list. In case you want to change the folder where the script files are you can change the path in the path text field. When you've written the path to the new folder, click Set Path and Refresh File List to have the files from the new folder show up in the File List.

---

1 You will have to reinstall the game if you do. That is why this is stated repeatedly.
4 Guide to modding The Sims 3

In this section I will describe the steps involved in modifying/improving my simulator. This guide contains info on how to modify items so they write to the log, how to make your own items, how to add items to the simulator home and how to ensure that the Master Program includes these new items to it's game state. All these steps deals with adding new items with sensors to the simulator.

If you want to add a computer that triggers a sensor when it's turned on to the simulation this involves three steps. The first step is modifying computers so they write script error files when they start and when they stop (section 4.1). The second step is adding a computer to the home in The Sims 3 (section 4.3.1). The third step is modifying the Master Program so it knows that it's supposed to keep track of whether the computer in the home is turned on and off (section 4.3.2).

If you want to add more script files to the simulator, see section 4.3.3.

A disclaimer: Do not update The Sims 3, that will ruin all the mods! Only update to 1.3.24, preferably use the file I've provided\(^2\).

4.1 Modding items to write to the log

Modding items is done by called core modding. Core modding consists of modifying the core files that makes up The Sims 3. You can read about how to do this in Appendix A: Core Modding! (For complete and total nincompoops, like me!). If you want to copy ILASM/ILDASM to a folder where you do your modding, do so in the IL-folder.

It's helpful, but not necessary, to read Appendix B: IL and DLL Overview/Tutorial.

In a nutshell, modding an item consists of finding where to write to a script error file and copying the right code at that place (more info on Script Error Files in section 4.1.1). Finding where is best done in .NET Reflector (which you can download online for free). Open Sims3GameplayObjects.dll here and find the class you want to modify. If the simulator has been installed according to the instructions you can find it in Development\ILFiles\DLLs. If you can't find it, also look through Sims3GameplaySystems.dll.

Once you've found your class look through the methods. You're looking for methods that can be considered start and stop functions. An example of this is MultiSeatObjects (sofas and the likes), these have two functions called StateMachineEnterAndSit() and StateMachineStandAndExit(). If you find two functions like this (and it doesn't have to be this obvious that they're start and stop functions) you're going to want to edit the start/stop function (section 4.1.2) and if not you'll want to look into editing the interactions (section 4.1.3).

\(^2\) I know I've said this before, but if you do an automatic update you will have to reinstall The Sims 3, which is why I try to make sure you can't miss this point.
4.1.1 Script Error Files

Data is brought out of the game through Script Error files. Script Error files are written by calling a function I've rewritten called ScriptErrorWindow. DisplayScriptError. It has two arguments. The first argument is a Proxy and the second argument is an exception. The exception is what will be written to the file, and I've decided to write it with the following format: DateTime.Now*objectID*on/off.

1) DateTime.Now is what you think it is, the current time. It's written on the format 04/20/2010 11:27:11 IE Date and time in hours:minutes:seconds. This is something you get with a separate function in The Sims 3.
2) objectID is an unique property that each item in The Sims 3 have. So with objectID you can separate between each single item in The Sims 3.
3) The third field will either be on or off depending on if the item just started being used or just stopped being used.

The stars is what I use to separate between the three fields. Simple spaces doesn't work because of the spaces in the time field.

Here is an example of how a call looks like in C#:

```csharp
Exception e = new Exception(DateTime.Now.ToString() + "*" + base.ObjectId.ToString() + "*on");
ScriptErrorWindow.DisplayScriptError(base.Proxy, e);
```

4.1.2 Editing the start/stop function

In this section you will find info on how to edit a start/stop function of a class. I am going to use Couches as an example to highlight how this is done.

Couches is an item type that inherits from MultiSeatObject, an abstract class for seatable objects with more than one seat. MultiSeatObject have two functions called StateMachineEnterAndSit() and StateMachineStandAndExit(). It's quite obvious what happens in these two functions: the first is called when a sim sits down in the couch and the latter when a sim rises.

First, find Couch in .NET Reflector. You will see that Couch inherits from MultiSeatObject (next to the name there is a link to MultiSeatObject). It's MultiSeatObject has the two mentioned functions. We start with StateMachineEnterAndSit() and looks what it does. The first thing that happens is this:

```csharp
if (!base.SittableStateMachineEnterAndSit(smc, sitPosture, routingSlot, sitContext))
{
    return false;
}
```

What happens here is that it returns false if it can't preform the base class EnterAndSit, which means that sitting down failed. Obviously it's a bad idea to write that someone sat down in our MultiSeatObject at this point.

Further on we see the following code:
SittableComponent.SitContext context = sitContext as SittableComponent.SitContext;
if (((context != null) && (context.PreferredSeat != null)) && (context.PreferredSeat.ContainedSim == null)) {
    entry.TargetSeat = context.PreferredSeat as Seat;
    if (entry.TargetSeat != null) {
        sitPosture.Sim.InteractionQueue.AddNext(entry);
    }
}
return true;

It does a check, does some things if said check was true and then it returns true. When it returns true someone has sat down, so that's where we want to write to the log. Open the code and search for the function (you can see if it's in GameplayObjects or GameplaySystems in Reflector by seeing where you found the class you want to modify). Press search→search and search for ClassName::FunctionName (note that there are two colons). In our case this is MultiSeatObject::StateMachineEnterAndSit. That search command will take you straight to the end comment of the function you want to find (the comment will say End of method Class::Function). If you don't get there straight away just press search→next until you are there.

Scroll up into the function (you're at the end of it so you need to scroll up to see the actual code). This is the IL representation of the C#-code you saw earlier in Reflector. What you want to find is where you want to add your call to DisplayScriptError (in our case right before the return true). Due to the code being in IL this can be a bit tricky. The intuitive thing is that the last ret call is the return true that's placed last in the C#-code. But note what is put on the stack just before the ret. It adds a zero, meaning that call returns false. If we go back to reflector (it's smart to have the function open in Reflector when you're working with the IL-code) we can see one other return call in the function, meaning the return false that happens on row three. So, the return that's placed last in the IL-code is in fact where the code jumps if the first if-statement returns true.

Just above that last ret is another ret. This ret pushes a 1 to the stack just before returning, meaning it returns true. Right before it pushes the 1 to the stack is where we want to place our code. Be aware that there's an if-statement just before the return true statement and if that if-statement fails the code is going to jump to the line where the 1 is pushed to the stack. So your code should go in between the label and the ldc.i4.1 line.

The code that should be added looks like this in IL:

call     valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()              
stloc.s  V_3
ldloca.s V_3
constrained. [mscorlib]System.DateTime
callvirt  instance string [mscorlib]System.Object::ToString()
ldstr      "*
ldarg.0
call     instance valuetype [SimIFace]Sims3.SimIFace.ObjectGuid
stloc.s  V_4
As you can see there's a load of pushing to the stack before calling new functions that pushes something new to the stack. This code can be copied straight off with two exceptions, namely the value of the sensor data and the variable names. The value of the sensor data is easy, if you are in the start function the bold line in the code should say 'on' and if you're in the stop function is should say 'off'.

On variable names: stloc.s V_3 is taking the first item on the stack and saves it to the variable V_3. It then pushes V_3 to the stack again. These variable names are associated with different types at the start of the function. The code is going to originally look something like this:

```
.locals init (class Sims3.Gameplay.ObjectComponents.SittableComponent/SitContext V_0,
```

We need three different variables in our code so we should add code so that it looks like this:

```
.locals init (class Sims3.Gameplay.ObjectComponents.SittableComponent/SitContext V_0,
             class Sims3.Gameplay.Objects.Seating.Scoot V_1,
             class Sims3.Gameplay.Objects.Seating.Scoot V_2,
             class [mscorlib]System.Exception V_2,
             valuetype [mscorlib]System.DateTime V_3,
             valuetype [SimIFace]Sims3.SimIFace.ObjectGuid V_4)
```

As you can see, we've added three fields: an exception, a DateTime and an ObjectGuid. These should be named so that they have names that don't collide with the old variables. For instance, a function with ten variables will already have variables called V_0, V_1 ... V_9, so V_10, V_11 and V_12 are fitting names. It's important to rewrite the error log writing code so that it reflects this.

Once you've done all this, save and assemble the new code. Open the resulting DLL in Reflector and find the function you changed. Check that your call is placed at the right place. If it is, copy it into the DLL fresh folder, go to The Sims 3\Mods\Packages folder, open the appropriate package file and import your new file. If an appropriate package isn't there you need to add your own. Check in Appendix A how to do this.
If you update one of these DLL's you will have to rebuild all my custom made items too, otherwise those will stop working. Check how to do this in section 4.2.1.

In short, to get this approach to work you have to:

1) Find the function you want to mod in .NET Reflector, then go to the corresponding IL-function.
2) Add the three variables to the list of local variables.
3) Give them appropriate names.
4) Find the correct place in the code to paste in our error writing code.
5) Change the variable names to the correct ones.
6) Make sure you're writing the correct thing to the log IE if you're in the stop function make sure you're writing off.
7) Save and assemble your new DLL.
8) Check in reflector that the code looks correct.
9) Copy your new DLL over the old one in the DLL folder.
10) Go to the mod folder in the Sims, go into the correct package for this DLL and reimport the DLL.
11) Rebuild all custom items.

4.1.3 Editing the interactions

In this section you will find info on how to edit an interaction of a class. Interactions are separate classes where The Sims 3 puts most of it's code for how items are used. This process is quite similar to the process described in section 4.1.2, so in this section you will only find the differences.

The most obvious difference is how you find the right function. You will have opened reflector and looked for a viable start and stop function but not find it. What you do next is that you open the OnStartup function of your class. In this function you will find a bunch of AddInteraction calls. Click on an interaction that's added and you will be taken to the class. This class has a run function, and it's in here you will want to add your code. You will want to add this code for each suitable interaction. Which interaction is suitable depends on what you want to do. If you want to add a sensor to computers that checks if they're on or off there's no use adding a call in the "repair" interaction.

Apart from finding where to add the code the difference is in the code itself. When writing in an interaction you'll have to go one step further to find the correct objectId and the correct proxy since you're referring to a different class than the one you're in. The new C# code should look like this:

```csharp
Exception e = new Exception(DateTime.Now.ToString() + "*" + base.Target.ObjectId.ToString() + "*on");
ScriptErrorWindow.DisplayScriptError(base.Target.Proxy, e);
```

Compare this to the code in section 4.1.1. The difference in the code is that you use base.Target instead of base to get the ID and proxy. This means you will have to find one extra thing when you add your code (you have to put the base object on the stack, at two different places). The new code looks like the following (important lines put in bold):

```csharp
call       valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
stloc.s    V_5
ldloca.s   V_5
constrained. [mscorlib]System.DateTime
```
Those two calls to ldfld is where you load base.target. This call looks different for all different objects (though, all interactions for one item have the same call since you're calling the same object), so you'll have to look around in the code to find how this call looks. You should look for another call to base.target and copy that code. If you can't find it, use any base.something call (base.actor is another popular one) and change the 'something' to Target.

### 4.2 Custom Made Items

In this section you will find info on how to make custom-made items. Custom-made items are good for when you want to add more behavior than simple sensors that say when the item is and isn't in use. One example of where custom-made items was made was for the sensors that sense how many sims are in a room. These sensors don't simply write on and off, they will instead write how many sims are in a room, and when this number change they will write the new number of sims.

Info on how to do this can be found in Appendix C: Object Modding (AKA adding interactions). You won't have to do the exporting of DLLs, those DLLs can be found in Development\ILFiles\DLLs. Also, in your C# project, set the build path to Development\ILFiles\Custom Items DLLs.
When you clone the object a good idea is to change the prize to 0 simeons before cloning it (we don't care about the gaming aspect of The Sims 3 so we might as well cheat until our faces turn blue).

The main advantage with cloning an item and adding your own code is that you don't have to bother with IL-code. You can instead write straight in C# (and C# is really simple to code in). The main disadvantage is that every time you make a change in your core DLL's you will have to rebuild your custom objects and import them into your package files, so the more custom objects there are in the simulator the more time it takes to get your new files into the game (it's not at all hard, it's just time consuming).

4.2.1 Rebuilding your Custom Items

If you've already done this before on this computer, simply open up the project and click f6 to build (Build→Build solution works too). Then go to The Sims 3\Mods\Packages, open the corresponding package file, find the entry with the S3SA tag (if you can't see the tag tab you will have to go to the furtherest left of the tag field and drag right), select and it and press grid. Then click the import/export arrow, press the import button and import your new DLL.

4.3 Remodeling the Sims 3 Home and adapting the server to the new home

In this section I will explain how you expand the simulator home and how you add new items to it. I will use an example of making a new room with a computer inside of it.

Before you start building our room it is nice to know what to do if you need more cash (everything in The Sims 3 costs simeons, except for the items we've modded that don't). First, click ctrl+shift+c to open the cheat console and type testingcheatsenabled true. Next, press ctrl+shift+c and type Motherlode (note the capital M). This will give you 50000 simeons.

4.3.1 Updating the simulated home

The first step is simply building a new room in The Sims 3. Go to build mode and build a new room in attachment to the old home. Build walls, add some nice floor and some nice wall covering (shift+click to fill the entire floor/wall with the same paint), also add some nice wall covering to the outside. This isn't really necessary for your room to work but a room without these looks ugly. Also add windows to the room and a door that connects the room to the rest of the house (the door is a requirement, without it your sims won't be able to enter the room).

Next, go to buy mode. Computers exists in the study, so click the study tab. You can see the computer icon, but you can also see a table icon and a chair icon. Your sim will want to sit by a table on a chair when using this computer, so first click the table and select a suiting one. Place a chair by the front of this table and last click a computer and place it facing the chair.

Also, you've made a new room. This means you should add an alarm that keeps track of how many sims are in the room. Go to kitchen and click the fire alarm icon. Take the first alarm on the resulting list (the one for 0 simeons) and place it somewhere on a wall in the room.
4.3.2 Updating the Master Program

Check that the Master Program is running and that it's set to Sims mode. Go to Live Mode, select your sim and click “play computer game” (or modded interaction of choice). This will (if you've constructed the room correctly) make the sim move into the room, sit down by the computer and start using the computer. Because someone's using the computer ScriptError files will be created, but since the Master Program is running they're going to be deleted and written to the log (placed in Documents\Electronic Arts\The Sims 3). When you enter the room you should be able to go to the log and see a sequence of script errors that looks something like this:

04/29/2010 11:27:11*0xc3ea00328b7fb170*0
04/29/2010 11:27:11*0xc3ea00328b7fb180*1

The date will vary, and so will the ID's. Take the ID of the item that turned to 1 and go to the GameState code. Go to the GameState constructor. There you will see a list of items being initialized. Copy one of the room items being added and add it last to the list. Change the ID of the copied field to the ID of the last entry in the log (the one that turned 1 when the sim entered the computer room) and change the name of the room to something fitting (in our computer example Computer Room should do it).

Next up is finding the ID of the computer, which is more tricky. When the sim starts the computer the computer will log something like this: 04/29/2010 11:27:11*0xc3ea00328b7fb170*on. When he stops using it the computer will log something like 04/29/2010 11:27:11*0xc3ea00328b7fb170*off. So when the sim has started using the computer, immediately click the computer icon in the top left corner to force the sim to stop using it. Wait for him to stop, then open the log again. There should be one item that's recently been both started and stopped in the log (after the entry of the sim entering the computer room). When you've pinpointed this item, close the log and tell the sim to turn the computer on again. Reopen the log, and you should see that this item has been turned on but not yet been turned off. If you've found it, add this item to the GameState constructor as well.

Once you've made your changes to the GameState you are going to have to rebuild the project. Open the command prompt, change the path to ..\Development\MasterProgramAndClientShell and type ant.

Last, make the sim stand up. Once he has, save the game. This should cement the computer's existence in the game.

4.3.3 Adding script files to the Master Program

Making a script file is done by copying the log entries created with the Sims 3 into it's own file. Apart from knowing which entry is the last one you only have to think about one thing: make sure the first entry in your script file is an entry where the sim enters the room. If you're making a “make dinner” file, the first script entry should be the one where Kitchen is set to 1. If you're making a “watch TV” entry and the Living Room is set to 1 a lot earlier than the sofa is activated you can simply copy the Living Room entry and change the time it's set to one a few seconds before the sofa is set to “on”.

The reason you have to do this is simply that in between the running of different script files all entries is set to off.
4.4 Hints and Tips

Visual C# is a great way for looking things up. You can quite easily go through all the fields of a class (say, sofas) in Visual C#. You can also write the code you want to write in IL in a project of your own. Just write the code you want to write, compile it and disassemble the dll and you have the code in IL. Be warned that sometimes it's not going to use proper stloc calls though (it's a fan of going for stloc.0 rather than stdloc.s Variable_name). So you'll still have to make sure the code is the correct one, but it's at least good help on the way to the goal.

4.5 Change log of functions

`ScriptErrorWindow.DisplayScriptError()`

Removed most of the code. Now all that happens is that LogScriptError is called and DisplayScriptError returns true.

`ScriptError.WriteError()`

Removed most of the writing to the file. Now all that is written is the exception.

`MultiSeatObject.StateMachineEnterAndSit()`

Added functionality that writes to the log.

`MultiSeatObject.StateMachineStandAndExit()`

Added functionality that writes to the log.

`Nap.Run()`

Added two calls to ScriptErrorWindow.DisplayScriptError.

`Fridge.PlayCloseSound()`

Added a call to ScriptErrorWindow.DisplayScriptError.

`Fridge.PlayOpenSound()`

Added a call to ScriptErrorWindow.DisplayScriptError.

`Stove.StartCooking()`

Added a call to ScriptErrorWindow.DisplayScriptError.

`Stove.SimStoppedCooking()`

Added a call to ScriptErrorWindow.DisplayScriptError.

`Toilet.UseToilet.Run()`
Added two calls to ScriptErrorWindow.DisplayScriptError. I couldn't add the call where I wanted to add it though. I wanted to add it within an if statement but I wasn't able to get the code to compile if I did that. By commenting out parts of the code I could compile it, and I eventually managed to compile all parts of the code separately, but doing it all at once failed.

Other interactions that might be worth checking out is clean and the upgrade interactions (tinker and UpgradeSelfCleaning).

Shower.TakeShower/CleanShower/Tinker.Run()

Added two calls to ScriptErrorWindow.DisplayScriptError in all these three classes run functions.

There are two more interactions worth checking out here: UpgradePreventBreakable and UpgradeSelfCleaning.

4.6 List of Modded Items

All sofas, all fridges, all stoves, all toilets, all showers all beds. There are also an alarm (the free one with the description text that says “keeps track of when someone enters and leaves the room”) that you can place in a room to keep track of how many people are in that room at every point in time.
Appendix A: Core Modding! (For complete and total nincompoops, like me!)
By BailaBaila99

So I've always had this dream of one day, having my own cooking show. The tagline would be: "I have no idea how to cook, so if I can do it, you can too!"

That's basically the idea behind this tutorial. I do not possess l33t skills like Pescado or Lemmy101 or Delphy. In fact, until yesterday, I had no idea how to core mod. But I bugged the hell out of the right people until they held my hand and walked me through it, and now, that's what I'm going to do for you, as a public service to those great modders, so that you won't have to bug them like I did. At least, maybe you'll bug them with slightly less stupid questions than mine. But enough introductions, let's get started. Oh, one more quick thing before that. This tutorial was not written by me. Well, it was, but not really. Really it was written a little bit by Delphy (whom I pestered in the chatrooms) and for the most part by Lemmy101. They showed me exactly what to do, I took notes, and now I'm passing it on to you. I'm just the middleman.

I. What is core modding?

A lot of you are probably familiar with XML tuning, a tutorial about which can be found right here on this very forum. A lot of you, also, probably know that XML tuning's fatal flaw is its ridiculous limitations! If you want to make some very minor adjustments to the way the game works, XML may be able to do it, and you should definitely check there first, but most everything else about the game is deeper coded in main DLL files. It turns out that even some very minor changes can only be made in these DLL's, but the plus side of that, is that the limitations are almost non-existent here. If you wanted to turn the Sims 3 into a first-person shooter set in a giant themepark with zombie clowns on the loose, well you might not be able to do that, but frankly I wouldn't be surprised. Editing these DLL's, however, is a kick in the head, but it's worth it if you want to mod the game, and even though it's a labor-intensive process, it actually ISN'T something far beyond the abilities of the average Joe Shmoe on this forum, thanks to the geniuses who figured it all out for us. Believe me, if I can do it, say it with me, you can too.

II. Ok, first of all, let's go over the laundry-list of programs you will need.

A. First, and foremost, you will need S3PE, which is in the early stages of becoming what SimPE was to TS2, if you don't already have it, get it here: S3PE (S3PI Demo Package Editor)

AA. Also, I hope this goes without saying, but you need the modding framework and Resource.cfg for your Sims 3 directory, in order to run any mod, and you MUST have the included d3dx9_31.dll file, in the correct directory, to run any core mod. If you don't already have 'em, get them here: Framework. (Note: if your game freezes during the initial loading screen with any core mod installed, it's because you don't have d3dx9_31.dll, or you don't have it in the right directory)

B. You will also need the latest .NET framework to be able to use the aforementioned program. If you already have it, move on to bullet C, if not, go here: .NET Framework AND you'll need the .NET sdk 2.0 for other stuff, so if you don't already have it, get it here: .NET SDK 2.0. And finally, (thanks to TigerM and Lemmy for pointing this out) you're definitely going to need the Windows SDK as well, if you don't already have it.

C. The .NET Reflector is a program that will give you a free peep-show of the DLL's you'll be modifying in a straight-forward intelligible format that I'm confident some species of monkeys
would have little trouble deciphering, but will not let you edit them (at least, not in the correct way, and not without wreaking havoc on your game). Why is this necessary, you ask? Because the DLL's are not made to be edited in their present state, after we convert them to a format we CAN edit, they will be much, much harder to read. Seriously, those monkeys from before, they will cry. Just trust me and get the program. **.NET Reflector**

D. Notepad can only handle so much abuse, and unfortunately it is ill-equipped to open the vast filesizes we need it to open. You'll need **Notepad++**.

E. ILASM and ILDASM: After you have installed all of the above, you will need to make use of two programs that should probably ALREADY be on your computer. The trouble is finding them, as they can be in a number of places. The first place to look, however, would be any of the many folders occupied by Microsoft's .NET in your system folders, i.e. Program Files and Windows. You will need to COPY (do not take them out, COPY them!) the exe files for these programs, and any dll's or other files they may need in order to run, to a new folder, anywhere on your computer, which is going to be your new modding workstation. Just make a folder in My Documents if you want and call it "MyStupidMod." Whatever floats your boat. My guess is that you'll need the same files I needed, which are: **ilasm.exe**, **ilasm.exe.config**, **fusion.dll** **ildasm.exe** and **ildasm.exe.config**.

Here's where I found mine, but yours may be somewhere different. If all else fails, try running a search of your entire computer to find the files. I found ilasm.exe, ilasm.exe.config, and fusion.dll in C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727, and I found ildasm.exe and ildasm.exe.config in C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin.

Again, I can't stress enough, COPY them instead of moving them.

F. Last though certainly not least, a statement riddled with irony that you will only come to contempuously appreciate once you actually begin to download and install this last program, is **Visual C# 2008 Express Edition**. But don't go and download it just yet! Listen, this is a very handy and free program that will allow you to create C# code, but it may not be entirely necessary for you. It is definitely necessary if you are looking to create a more in-depth, larger scale mod, like the Indie Stone mod. For my mod, however, I only needed to change a single value, and I didn't open this program once in the process. Bascially, if you want to create new DLL's, you're going to need this program (or you can choose to do a pretty advanced workaround described by Rick and TigerM below). If you just want to modify the existing DLL's, skip this step, for sanity's sake, and move on. The only real reason for the hesitation here, is that this program takes a LONG ASS TIME to install. If you do end up installing it, make sure you have a good book. Or the firefox Stumble! add-on. Bascially, you're gonna have some time to kill.

That's it! Once you've gone through all of these steps, you are finally set up to core-mod. Now we can get started. (Didn't I tell you this was labor-intensive?)

III. Preparing the files for editing

A. First things first, open S3PE, and you'll need to open the right package and extract the right file. For your specific case, I don't know what you want to do here. You might want to make a fresh, new mod for the Sims 3 base game, or you might want to add some small detail to an existing mod that you can't live without. (note that if you want to keep your Indie Stone or Awesome Mod, you must make your changes to THESE mods, due to the highlander rule, there can only be one. This wouldn't be true of XML tuning, but we're in Core Modding country now, and things aren't quite as simple (yet)) There are 3 main DLL's that control nearly every aspect of the game. These are:
Sims3GameplaySystems.dll (likely the one you're going to want to edit), Sims3GameplayObjects.dll, and UI.dll. Both Awesome Mod and Indie Stone modify all three, (which is why the are incompatible, and would be even if they only modified one in common).

So, if you want to edit the basegame, open ProgramFiles\Electronic Arts\Sims 3\Game\Bin\Gameplay.package (NOT GameplayData.package, that was for XML). If you want to edit an existing mod, open that mod's package. Either way, you will see a short list of files (anywhere between 1 and 4 of them), and you need to pick the correct one to edit. To do so, select each, one at a time, and press "Value". In the window that pops up, you should see a line that says "ManifestModule:" and after this will be the original filename (it should have .dll at the end). When you've found the one you want to have a go at, make sure it's still selected, and click "Grid", then Import/Export, Export, and navigate to the directory we set up earlier as your workstation. Now, save it as the filename you saw earlier, when you checked the Value. For instance, if you're exporting Sims3GameplaySystems.dll, you'll want to save it as that. And don't forget the .dll!

B. Now, open the .NET Reflector program, and use it to open the dll you just exported. (NOTE: If you have trouble opening the file, it is likely because you don't have it in the same folder as ILASM and ILDASM. If you do, then something is wrong with these programs, or your file. Go back and make sure you followed these steps correctly.) You'll have to click some plus signs in the left window once you open it, to see what's inside, but what you'll find is a very handy display of the entire contents of the file. Take this opportunity to do some detective work. Once you find an item that looks like what you probably want to edit, double-click it, and you'll see its contents in the right window. My goal was to make a "No Privacy" mod, so I went to Sims3GameplaySystems > Sims3.Gameplay.Situations > GenericPrivacySituation > OtherSimCanIgnorePrivacy(Sim) : Boolean, and found the following:

Code:

```csharp
public override bool OtherSimCanIgnorePrivacy(Sim sim)
{
    return sim.SimDescription.ToddlerOrBelow;
}
```

Now, the syntax might be scary, but hopefully a 5-year-old wouldn't have too much trouble ascertaining what's going on here. The condition, OtherSimCanIgnorePrivacy, is checking two factors, 1: is the invading Sim a spouse? and 2: Is the invading Sim a toddler or baby? If either of these factors is true, the condition returns true (and the sim is allowed to ignore privacy in the situation) but otherwise, it returns false (and the sim pitches a fit, gets embarrassed, and no one ends up using that bathroom). Well, I wanted to skip these two checks entirely, and simply return "true."

If we could edit this RIGHT NOW, in a mostly legible fashion, the world would truly be filled with sunshine and puppy dogs, but sadly it's a cruel, cruel place, and we can only view the code here. On to C.

C. We must convert our little DLL into a behemoth of a file of another type, called IL. The code will all be converted, so that we can edit it, into a much less user-friendly (if not a downright user-asshole) format, the filesize will be somewhere around 50mb. To do this, we need ILDASM, one of those strangely named programs we put in our workstation folder earlier. (REMEMBER: ILDASM
is for IL Disassembling, and ILASM is for IL Assembling. Basically, ILDASM = DLL>IL, ILASM = IL>DLL. Don't go and double-click it, that won't work. Here's what you do:

Go to start > run, and type "cmd". This will bring up a DOS console, and you'll need to navigate to the directory we're working in. To do this in DOS, type "cd directoryname" where directoryname is the name of the folder you're navigating to, in My Documents. If you need to go up a folder, type "cd..". If you're having trouble with this, go here. Once you've gotten to your workstation directory, type the following (NOTE: syntax can be tricky):

Code:

```
ildasm filename.dll /output=filename.il
```

I'm hoping you already figured out what you're supposed to type in place of "filename." If all goes well, a window should appear and a progress bar should begin traversing a small window on your screen, and when it finishes you will have an IL file in your workstation directory.

IV. Making your Edit! (FINALLY!)

Now you'll want to use Notepad++ to open the IL file you just created. Take a moment to recover from the shock of how lengthy this file actually is. Go ahead, I'll wait for you. Ok, traversing the breadth of this file is going to be a headache, so you'll want to use the search option a lot. (NOTE: The search box has an up/down toggle. Just so ya know.) Start by searching for some of the terms you found when you did your detective work in the .NET Reflector earlier. If you don't remember them, you can re-open it and check, the dll is still there. These terms will likely occur multiple times in the document, but if you pick something really specific to search for, you won't have to do quite as much hunting. I searched for "CanIgnorePrivacy" and found a few dead-ends, but finally came across this (the definition of the method I was looking for):

Code:

```
.method public hidebysig virtual instance bool OtherSimCanIgnorePrivacy(class Sims3.Gameplay.Actors.Sim sim) cil managed
{
  // Code size 12 (0xc)
  .maxstack 8
  IL_0000: ldarg.1
  IL_0001: callvirt instance class Sims3.Gameplay.CAS.SimDescription Sims3.Gameplay.Actors.Sim::get_SimDescription()
  IL_0006: callvirt instance bool Sims3.Gameplay.CAS.SimDescription::get_ToddlerOrBelow()
  IL_000b: ret
} // end of method GenericPrivacySituation::OtherSimCanIgnorePrivacy
```
This is MSIL code, and if you don't know how to edit it, well, neither do I. You can read up on it, on the internet, I'm told, and you can check out this helpful topic by Lemmy101, {UPDATE: Also check out Lemmy's New IL / DLL Tutorial} but really, this is where the real difficulty, and creativity comes into play.

I can tell you, however that those strange bits like "IL_0001" are the line numbers inside the {curly brackets}, and that there cannot be two with the same name inside the same set of brackets.

Now, I probably would have had to play around with this file and think really hard for days until I figured out how to make the simple change of defining this condition as simply "true," but Lemmy101 geniusly pointed out that we could just copy and paste from the method ABOVE this one, which happened to be the following:

Code:

```
.method public hidebysig virtual instance bool OtherSimCanJoinSituation(class Sims3.Gameplay.Actors.Sim otherSim) cil managed
{
    // Code size       2 (0x2)
    .maxstack  8
    IL_0000:  ldc.i4.0
    IL_0001:  ret
} // end of method GenericPrivacySituation::OtherSimCanJoinSituation
```

The condition OtherSimCanJoinSituation for GenericPrivacySituation is simply returning "false." We want to the CanIgnorePrivacy method to return "true." So, we copy and paste, with one little change. Apparently, the item that's really doing the work here is the unassuming "ldc.i4.0". I have no idea what the hell this means, but lemmy told me to change it to a .1 (for true) instead of a .0 (for false). Here's what our new code looked like:

Code:

```
.method public hidebysig virtual instance bool OtherSimCanIgnorePrivacy(class Sims3.Gameplay.Actors.Sim sim) cil managed
{
    // Code size       12 (0xc)
    .maxstack  8
    IL_0000:  ldc.i4.1
    IL_0001:  ret
} // end of method GenericPrivacySituation::OtherSimCanIgnorePrivacy
```
Then save, and that's that. On to V.

V. Un-preparing the file and sticking it in your game.

A. First things first, we need to get our edited IL file back into DLL form. Before you do, though, consider that you might not want to replace your original DLL you exported way back in an earlier step that I'm too lazy to look up the name of. If you don't want to replace it, rename it something silly, and it'll be left alone. If you just don't care, the let it be. Now go back to that cmd DOS window. In your workstation directory, enter the following, using ILASM this time, and again remember that the syntax is tricky:

Code:

```
ilasm filename.il /dll
```

You should see a vortex of text come careening down the window at you. It will do this for a good 30 seconds though, so you can take this opportunity to pretend you're Neo from the matrix.

If all goes well, (and this is the part you have to cross your fingers for) it will finish up and say "Operation Completed Successfully" and you'll have a newly modded .dll file in your directory. Whatever the name of this dll is, make sure to change it (if need be) to that original filename we saw in S3PE > Value at the start of this whole mess.

...If it FAILS, then either you didn't type it in the DOS window correctly (and you're hoping for that possibility, trust me) or your mod code doesn't make sense, and it couldn't compile it. In which case, you'll have to figure out what you did wrong back in Notepad++ and try again. (and again and again and again)

B. Once you've got your DLL file finished ILASM'ing, go ahead and open it with Reflector. Check the part or parts you modified, and see if you managed to do what you were trying to. Did it work?

C. If so, open S3PE, and depending on whether you're editing an existing mod, or creating your own new base-game mod, either open the mod package, or simply create a new one.

If you're going with door number one, then all you'll have to do is find that original file you exported before in the mod package (making sure it's the same one by clicking "Value") and simply select it, then Grid > Import and choose the DLL that's still warm from the ILASM. When it asks you if you want to commit the change, say yes, and then save the package and put it in your ProgramFiles\ElectronicArts\Sims3\mods directory (making sure there's no other core mod packages already in there, as they will almost certainly conflict).

If you're more of a number two person, then you'll need to re-open the original gameplay.package (in a new window of S3PE, for your convenience) and find the file you exported before. Now, in the other S3PE window, your new package window (DO NOT ADD ANYTHING TO THE ORIGINAL GAMEPLAY.PACKAGE FILE! Your sims will all die horrible deaths and so will
your computer.) go to Resource > Add Resource, and copy in the Type and Instance from the file in gameplay.package EXACTLY. The group should simply be "0x00000000" (that's zero x and 8 zeros). After you've added the resource, select it, and Grid > Import, choose the ILASMed DLL and commit, then save, put the package in your mods directory, making sure there's no other core mods present (as they will almost certainly conflict) and you're done!

// End of Tutorial *whew*

Anyway, I wish you all the best of luck with your modding, and I can't wait to see what you create. Note that during the majority of the steps in this tutorial I had no idea what the hell I was doing, and if your results turn out different than mine in any area, I will try my best to offer help, but it's more than possible that I'll be of no more use to you at that point.

Also, if anyone thinks they can improve or expand this tutorial, I have only this to say. OH SO YOU THINK YOU CAN DO IT BETTER, HUH? No, really, that would be awesome. Any flaws you can point out on my part, or any extra guidance you can offer in the MSIL editing part (which deserves a whole other tutorial) would be welcome.

That's it, thanks again to Lemmy101, and good luck to you all!
Appendix B: IL and DLL overview / tutorial

By lemmy101

First versions I'll elaborate on the different parts in more detail as I go on.

When writing core-mods, you will unavoidably have to delve into the world of Intermediate Language (IL) at some point or other. This is not a tutorial on setting up a DLL, or editing packages, but some basics on the IL language and how to read it and use it alongside the awesome Reflector to do all your modding.

Depending on how complicated your mod is going to be, your IL editing could be simply adding basic calls or removing the contents of methods, or changing the visibility of a class or method so it can be accessed from outside. More complicated use of IL may see you changing the execution path of the code, such as adding if statements, to make existing methods work differently.

This tutorial doesn't go into the specifics of ILDASM, ILASM, editing packages and whatnot. For a tutorial on these processes, check out the cool tutorial: http://www.modthesims.info/showthread.php?t=354419 by BailaBaila99. This tutorial will assume you can recreate those steps.

Differences between C# and IL

The main difference between C# and IL is the syntax. You'll notice when you first look at a bunch of IL code is it's a lot less immediately obvious what it is doing. There's no nesting with high level syntax such as 'for loops', 'while loops' or 'switch statements', instead everything is expressed as a long list of abbreviated instructions few if none of which are immediately obvious what they are doing.

Take this for example, which is the GetSide method inside the Door class in the Sims3GameplaySystems.dll, first of all in C#:

Code:

```csharp
public tSide GetSide(uint side)
{
    if (side == 0)
    {
        return tSide.Front;
    }
    return tSide.Back;
}
```

3 Appendix A: Core Modding! (For complete and total nincompoops, like me!)
So this is a method that takes in a number (side) and if that number equals 0, then it returns Front, otherwise it returns Back.

Now in C#, since we have the braces to nest blocks of code together, we can have an if statement that, if the result is true (side == 0) then the game will execute the code inside the braces. Otherwise, if the side == 1 or more, then the game will jump to the closing brace and continue execution to return the 'Back' value.

In IL, there are no such things as braces within a method, and the code is on a much lower level, relying on slightly obscure inline instructions. Therefore it looks a little less clear what's happening:

Code:
```
    GetSide(uint32 side) cil managed
{
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.1
    IL_0001:  brtrue.s   IL_0005
    IL_0003:  ldc.i4.0
    IL_0004:  ret
    IL_0005:  ldc.i4.1
    IL_0006:  ret
} // end of method Door::GetSide
```

A little more scary than C#, but we'll dissect what it's doing:

Code:
```
    GetSide(uint32 side) cil managed
{
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.1
    IL_0001:  brtrue.s   IL_0005
    IL_0003:  ldc.i4.0
    IL_0004:  ret
    IL_0005:  ldc.i4.1
    IL_0006:  ret
} // end of method Door::GetSide
```

The above code is the method's definition, just like:

Code:
```
public tSide GetSide(uint side)
```
Just in a little more verbose form, it's giving all the same information there, so we'll just take it that these are equivalent.

Then there is the open braces for the method (though no braces allowed inside here)
Ignoring comments ( lines beginning with // ) as they have no impact on the generated dll.

And then the following line:

Code:

```
.maxstack 8
```

This line is specifying how big the stack is.

**The Stack**

The stack, in case you're wondering, is like a list of things (number, objects, whatever) that you are using at a time. Though unlike a list, a stack behaves like... well, a stack of objects. So when you add something to the stack, it is piled on the top of it. 'Push' another thing to the stack and that is piled on top of that. Then the first time you take an object off the stack, it is taken off the top, so you'll get the last object you put on the stack back off of it. 'Pop' another off the stack and you'll get the second last object you placed. Keep going and last of all you'll take the first object you put on there.

So you 'push' objects onto the top of the stack, and 'pop' objects off the top of the stack.
In this case we have 8 'slots' on the stack, so we can push 8 objects on before having to 'pop' any off.

It's unlikely you'll need to worry too much about this.

**The IL Code**

If we look at the first line of the actual IL code we can see how it's structured:

Code:

```
IL_0000:  ldarg.1
```

IL_0000: specifies a label for a line, so it can be referenced in other parts of the method, and can be anything you want (without a space, I assume, though I've not tried with a space). If you were to write some new IL code from scratch, you would only tend to have a label on a line of code of some importance, where you may want to 'jump' to from another section of code. But since this dll wasn't written in IL in the first place, but rather compiled in Visual C#, it doesn't have sensibly named labels and as such when you ildasm the DLL, it will simply label EVERY line with a default label in the format IL_XXXX (where XXXX is a number that I assume is related to the offset of the instructions, though the only important thing is they are all different)

So what does this instruction do? Well ignoring the IL_0000: label, the next thing is 'ldarg.1'. This is a 'load' instruction, as you can tell by the fact it starts with 'ld'.
A load instruction loads 'something' and plops it on top of the 'evaluation stack'. Remember that a stack is a 'last on, first off' pile of things, so if you need to use a value, it needs to be ON TOP of the stack.

The above 'load' is a 'load argument' or 'ldarg'. An argument is the variables passed into a method. To go back to the C# code:

Code:

```csharp
public tSide GetSide(uint side)
```

As we can see, there is one argument, which is a uint number representing the 'side' that the method will use to determine which of the two return values it uses.

Thing is, this method is inside a class, therefore objects in the game that are of this class will have this method that others can call on it. In IL we do not have 'this' identifier to reference the object that the method is part of. In C# you could do 'this.Something()' inside a method to reference another method or variable on the instance that you're calling the method on. If this makes no sense then you'll need to read up a bit on C# on how instances of classes work.

So anyway, such a thing as 'this' doesn't exist in IL, so instead we treat the method like it's got another invisible first argument that's not present in the C# code:

Code:

```csharp
public tSide GetSide(Door this, uint side)
```

So argument 0 (we always count from 0) is 'this door' and argument 1 is the 'side' unsigned integer number that would actually be argument 0 in the C# code, which uses 'this' instead of adding another argument.

So... from all this we can determine that the line:

Code:

```csharp
IL_0000: ldarg.1
```

Is loading the number value of 'side' to the 'top of the evaluation stack'.

Next line is as follows:

Code:

```csharp
IL_0001: brtrue.s IL_0005
```

A 'br' is a jump instruction. A 'br' instruction will jump to another part of the IL method. After a br instruction will be the target label that the br should jump to. There are different br instructions, the two most common ones used being 'brfalse' and 'brtrue' which will jump if the thing on the top of
the evaluation stack is 'false' or 'true' respectively (The .s means 'short-form' though I'm not entirely sure what the relevance of this is yet. :D) If the check fails, instead of jumping to the target instruction, it will carry on on the next line. 'br' is another command that will ALWAYS jump regardless of what is on the evaluation stack.
It is using all these that we can have IL code that does checks and has numerous paths of execution.

So in the above two lines:

Code:

```
IL_0000:  ldarg.1
IL_0001:  brtrue.s  IL_0005
```

The first line loads the value of 'side' onto the top of the evaluation stack, and the second line sees if what the thing on top of the evaluation stack is 'true' and if so, will jump to the IL_0005 instruction. In programming, any non-zero number is classed as true, and 0 is classed as false. So in the above case, if 1 was passed as the 'side', then the game would jump to IL_0005. If the 'side' equals 0, then it would carry on on the next line of IL_0003 as such:

Code:

```
IL_0003:  ldc.i4.0
```

This ld command loads a number between 0 and 8 onto the top of the evaluation stack. In the above case it's loading 0 onto the evaluation stack... then...

Code:

```
IL_0004:  ret
```

This 'ret' command simply returns out of the method. Since this method returns a value (of the type 'tSide') which looks like this in C# code:

Code:

```
public enum tSide : uint
{
    Back = 1,
    Front = 0
}
```

The return returns the thing on top of the evaluation stack as the tSide that this method returns, in this case it has returned a 0, and so has returned tSide.Front.
If the 'side' value passed in was 1 instead of 0, then the call to:
would instead be putting a 1 at the top of the evaluation stack, therefore the following 'br' jump line:

would instead of carrying on at the next line, make the jump to IL_0005, skipping the code that returned the Front and running the following two instructions:

The first one loads 1 onto the top of the evaluation stack, and the next line returns, returning 1 or 'Back' to the calling method.

**Other IL instructions**

**Loads**

Another two useful 'load' instructions are ldfld and ldsfld.

These two methods load a field from inside a class onto the evaluation stack. From example:

In C#, if you had a Bob variable called 'myBob' you would access 'field' by calling:

Or if you were inside one of Bob's methods, you could access the 'field' by using:
In IL we access fields inside objects using ldfld.

To get at the 'field' variable inside the Bob class, we first need to get the Bob we want the field from onto the top of the evaluation stack. The Bob may be passed as an argument, or we may already be inside one of Bob's methods. Whatever way, we need to get him onto the evaluation stack, so if he was passed as the first argument into the method we would do:

**Code:**

```
ldarg.1
```

If we are inside the Bob we want the field from we would do the equivalent of 'this':

**Code:**

```
ldarg.0
```

Now, either way, we have our Bob that we need the field from on the top of the evaluation stack. Then we can call:

**Code:**

```
ldfld   int Bob::field
```

And now the value of 'field' of this particular Bob should be at the top of the evaluation stack. This would generate a dodgy DLL if a Bob isn't on top of the evaluation stack when the ldfld is called, and likely it would generate a crash if the game hit that code.

**Code:**

```
ldsfld  int Bob::field
```

You would also call the above ldsfld method in cases where the field is declared as 'static', which means it is a single value stored inside the class, and not a separate value for each object of that type:

**Code:**

```
public class Bob
{
    public static int field;
}
```
In the non-static version if you have 10 Bobs, each would have their own 'field' value, but in the static version all Bob's share the same 'field' value. In these cases you don't need a Bob on top of the evaluation stack for ldsfld, as there is only one field value we're interested in. So just call ldsfld without needing to push a Bob onto the evaluation stack first.

Rounding off the 'load' instructions, there is also 'ldloc', which we will cover in a later tutorial.

**Calls**

The most common IL instructions you will likely be adding. A 'call' will call a method on an object. There are two flavours:

Code:
```
call
```

And...

Code:
```
callvirt
```

The callvirt is a little more complicated so we'll leave that for now, but generally you won't have to worry about them.

A call can be used to call a static method inside your DLL. For example, if you download Visual C# Express and create a new 'Windows Class Library' project called MyMod and added the following class to it:

Code:
```
public namespace MyMod
{
    public class MyClass
    {
        public static void Initialise()
        {
        }
    }
}
```

If you create the above class in your DLL in Visual C#, and compile, you will then have this
method exposed so that your Sims3GameplaySystems classes could call it.

To do this you first need to add a reference to your freshly ILDASM'ed Sims3GameplaySystems.il code.

At the top of the file, you'll notice a section with a bunch of .assembly bits at the top like this:

Code:
```
.assembly extern SimIFace
{
 .ver 1:0:0:50
}
```

These specify a link or 'reference' between this DLL and other DLLs, so you can access the classes inside those DLLs.

We'll need to add a reference to our mod in there. In the above example where the name of the mod DLL and main namespace is MyMod we would add the following code:

Code:
```
.assembly extern MyMod
{
 .ver 1:0:0:0
}
```

If you change the version of the mod in your C# DLL project then you will need to update it here. Most people just leave it as 1.0 as the two files will always be distributed together and you can assure they're both for the same version, so you can leave this alone.

So save, and there you have it. Provided your MyMod.dll is present in a package The Sims 3 can see, this Sims3GameplaySystems.DLL can access all the classes in your mod.

So now we've done this, open up the Sims3GameplaySystems.DLL in Reflector and have a look around for somewhere to add a hook into your MyClass.Initialise(). We're looking for somewhere that's called once when you load a saved world, after the world has loaded. The place we use is:

Code:
```
Sims3.Gameplay.InWorldState.Startup()
```

So search for that in Reflector by opening out the various namespaces / classes until you find the Startup() method. You'll see the following code:
public override void Startup()
{
    base.Startup();
    Responder.GameStartup();
    if (World.IsEditInGameFromWBMode())
    {
    }
    MapTagWin.LoadMapTagLayout();
    this.mStateMachine = StateMachine.Create(1, "InWorld");
    this.mSubStates[0] = new LiveModeState();
    this.mSubStates[1] = new BuildModeState();
    this.mSubStates[2] = new BuyModeState();
    this.mSubStates[6] = new ShoppingModeState();
    this.mSubStates[3] = new CASFullModeState();
    this.mSubStates[4] = new CASDresserModeState();
    this.mSubStates[5] = new CASMirrorModeState();
    this.mSubStates[9] = new PlayFlowState();
    this.mSubStates[8] = new EditTownState();
    foreach (InWorldSubState state in this.mSubStates)
    {
        this.mStateMachine.AddState(state);
    }
   StateMachineManager.AddMachine(this.mStateMachine);
    this.mPostWorldInitializers = new Initializers("PostWorldInitializers", this);
    this.mPostWorldInitializers.Initialize();
    Gameflow.GameSpeed pause = Gameflow.GameSpeed.Pause;
    bool flag = false;
    if (GameStates.StartupState == SubState.LiveMode)
    {
        flag = (GameStates.ForceStateChange || !PlayFlowModel.Singleton.GameEntryLive)
            || (PlumbBob.SelectedActor != null);
    }
if (flag)
{
    if (Gameflow.sGameLoadedFromWorldFile)
    {
        pause = Gameflow.GameSpeed.Normal;
    }
    else
    {
        pause = Gameflow.sPersistedGameSpeed;
    }
}
Gameflow.sGameLoadedFromWorldFile = false;
string s = CommandLine.FindSwitch("speed");
if (s != null)
{
    int num;
    if (int.TryParse(s, out num))
    {
        pause = (Gameflow.GameSpeed) num;
    }
    else
    {
        ParserFunctions.TryParseEnum<Gameflow.GameSpeed>(s, out pause,
            Gameflow.GameSpeed.Normal);
    }
}
Gameflow.SetGameSpeed(pause, Gameflow.SetGameSpeedContext.GameStates);
NotificationManager.Load();
GameUtils.EnableSceneDraw(true);
LoadingScreenController.Unload();
switch (GameStates.StartupState)
We need to add a call to our 'Initialise' method from inside this method. It's a static method so we don't need a MyMod object with which to call it on, so the call will be quite simple. The best place to add it is after it unloads the loading screen, on the line:
So we do a search in the IL code with Ctrl-F in Notepad++ for LoadingScreenController and we see the following code:

```csharp
LoadingScreenController.Unload();
```

```csharp
Code:
```
IL_016f:  call       void [UI]Sims3.UI.NotificationManager::Load()
IL_0174:  ldc.i4.1
IL_0175:  call       void [SimIFace]Sims3.SimIFace.GameUtils::EnableSceneDraw(bool)
IL_017a:  call       void [UI]Sims3.UI.LoadingScreenController::Unload()
             Sims3.Gameplay.GameStates::get_StartupState()
IL_0189:  stloc.s    V_7
IL_018b:  ldloc.s    V_7
IL_018d:  ldc.i4.0
```

Note that your label number (IL_BLAH) may be different to mine, but that's not important for this example.

So we want to add the call after the LoadingScreenController::Unload() call, you may want to be careful when placing something after a call, as if the call has a return type (i.e. Not void like the Unload call) then the line after may store the return value. Storing (instructions beginning with st) are another important element of IL I'll explain in a follow-up tutorial. In fact that's exactly what's happening in the call immediately after it. The get_StartupState() is returning a 'Substate' and that is being stored in V_7. Exactly what V_7 means is outside the scope of this tutorial, just know if you wanted to add this call after that one, you would put it after the 'stloc' instruction, not after the call itself.

But we're placing ours after the LoadingScreenController::Unload() so we will copy and paste that line to make a copy of it directly on the line below, as such:

```csharp
Code:
```
IL_017a:  call       void [UI]Sims3.UI.LoadingScreenController::Unload()
IL_017a:  call       void [UI]Sims3.UI.LoadingScreenController::Unload()
             Sims3.Gameplay.GameStates::get_StartupState()
IL_0189:  stloc.s    V_7
```
Remember that no two IL instructions inside a single method can have the same IL label, so change our new line (the second one) to have a different label:

Code:

```
IL_017a:  call       void [UI]Sims3.UI.LoadingScreenController::Unload()
MYMOD01:  call       void [UI]Sims3.UI.LoadingScreenController::Unload()
IL_0189:  stloc.s    V_7
```

The call we've copied is accessing a class inside the UI.DLL. That is what the square brackets [UI] means. So we need to change this to point inside our mod, which is called MyMod:

Code:

```
MYMOD01:  call       void [MyMod]Sims3.UI.LoadingScreenController::Unload()
```

To access our class, we first write every namespace that the class is inside, seperated by a dot. In our case it is a single namespace called MyMod, so we do MyMod.MyClass...

Code:

```
MYMOD01:  call       void [MyMod]MyMod.MyClass::Unload()
```

Finally it's not Unload we want to call, but Initialise

Code:

```
MYMOD01:  call       void [MyMod]MyMod.MyClass::Initialise()
```

And that's it! If you ILASM the Sims3GameplaySystems.dll and package it with your mod, using the right instance Id for the core dll and a unique instance for your own dll, The Sims 3 should call this method the first moment after the loading screen is unloaded after the town is loaded.

Now using Visual C#'s intellisense, you can import Sims3 namespaces using:

Code:

```
#using Sims3.
```

And should be able to poke about in C# code at any aspect of Sims3 gameplay logic.

A useful thing to do in the initialise is this:
Code:

```csharp
mTimer = AlarmManager.Global.AddAlarmRepeating(StartTime,
    TimeUnit.Minutes, new AlarmTimerCallback(OnTimer), Frequency,
    TimeUnit.Minutes, "MyMod", AlarmType.NeverPersisted, null);
```

Where `mTimer` is an 'AlarmHandle' stored in your class, and `StartTime` and `Frequency` are the amount of time in in-game minutes between updates of your mod.

Then all you need to do is add the method:

Code:

```csharp
public static void OnTimer()
{
}
```

And when your mod is first initialised on town load, it will start a timer that will call the `OnTimer` method every 'Frequency' in-game minutes. The first update being 'StartTime' minutes of game-time.

Now you have your mod initialised and running alongside game-time as frequently as you'd like, so you can poke about with what you like, add new hooks on game events in IL, or adding 'listeners' in your code that are informed of Sims 3 events and respond to them.

For figuring out all these things, the best way is to nose about at existing mods and see how they did something.

Good hunting!
Appendix C: Object Modding (aka adding interactions)

By Kolipoki

Object Modding is not a very well-defined category. Usually an object mod is one that will clone an object and give it new interactions for it to be able to run. This tutorial will show you how you would set up to create an object mod, and add an interaction to it. By all means interactions are not all you can do with it. With coding experience you tell it to do a lot of things.

What You Need:

1. Microsoft Visual Studip C# 2008 Express Edition (http://www.microsoft.com/express/vcsharp/) : You will use this for making the your new object do what you want it to do instead of just being another cloned object.

3. Sims 3 Object Cloner (find this in Development\DevelopmentSoftware\InstallationFiles)

4. Sims 3 Package Editor (already installed)

5. The Net Reflector : This will be used to explore through the Sims 3’s code content.

6. Knowledge of C# Code: If you have no knowledge of C# code then you will be totally lost some time in this tutorial and will have to start asking questions about everything in your code. I highly advise you to read up on C# before starting this tutorial. (I know from personal experience). This is a pretty good site but it will not teach you everything: http://www.csharp-station.com/Tutorial.aspx (if you know Java that might be enough.)

Setting Up:

It’s best when making mods, to create a folder for all of your things. The best place is on the desktop, its easy to reference. After you have your folder you will need to obtain some main Sims 3 dlls that will be used as references later in coding. If you have all ready core modding then you will know what to do.

1. Open the S3pe
2. Hit - File - Open
3. Navigate to your Sims 3 main directory folder (usually in the c drive)
4. From there go to the Game Folder and then the Bin Folder
5. You will see 3 packages. These packages contain the dlls that you will be needing.
6. First open the gameplay package
7. Four files will appear, you will be exporting 3 of them.
8. Click on the first sting and hit value. Scroll down the lot of words untill you see Manifest Module:
9. If the Manifest Module says TestGameplay, you can close out of the window and continue to the next item on the list. If it doesnt say TestGameplay close out of the the window and hit "Grid". At the top of the data grid should be called Assemly. His the down arrow next to the Import/Export and hit "Export"
10. S3pe will then prompt you to save. Navigate to your mod folder and save the file as what the Manifest Module said followed by ".dll"
11. After you have saved, hit cancel in the datagrid and move on to the next item in the package.
12. Extract it the same way you did the first one naming it with the Manifest Module it has followed by ".dll"
13. Do these for all of the items in that package.
14. When done with that pick open up one of the other 3 packages in the Game/Bin directory and do the same. (be sure not to take the test gameplay, it is not needed)
15. When finished with that package move on to the last package and do the same.

Making you Object

The next step in this process is to make your new cloned object. Start by opening the S3oc (Sims 3 Object Cloner). First time users will have to navigate to their Sims 3 folder to start off the program. Once opened we will want to make our new clone so hit Cloning - and choose the type of object you will be using. For this tutorial we are using Normal Objects. S3oc will then load the object catalog with all the objects in the game. Next you need to choose witch object you want to clone. In this example i'm going to use the Teddy Bear witch is called ToyStuffedAnimalTeddyBear. When you have picked out your object hit the "Clone or Fix" button. S3oc will then ask you to determine some options. You will need to make sure you give it a unique name. We will name ours "Kolipoki_AddIntersectiontoBear". At this time you might also want to give it a new name, catalog description, price, and even which categories it will go into. Then hit start, and S3oc will then create your new object and ask you to save it to a new location.

Congratulations you have your own cloned object.

Setting Up MVSC# (Microsoft Visual Studio C#)

Before we start coding we need to set up MVSC# So that we can use it and make the code work. Of course there is all ready a tutorial on that out there.
http://www.sims2wiki.info/wiki.php?... Studio_project. Its small but you need to follow it. The files in step 4 are the files that we extracted in the beginning of this tutorial.

Finding the Original Object's Name

This next step is important because it will later be used to make your script be derived from the previous object making it the same so it will work in game. Start off by open your cloned object in the S3pe (in my case Kolipoki_AddInteractontoTeddy). Next your going want to make sure that names and tags are on. Click the buttons right next to them as shown in the screen shot. Afterwards find the item in the list with a tag name of OBJK. Open the OBJK with the grid button.

You should see a "Component Data" category click on the category and a small button with three little dots appears to the far right. Click on this button and it will bring up a new menu. On the left of that new pop-up, you should see two items. Click on the second item labeled "[1]CDTString". The data row should have a rather long name usually starting with "Sims3...." That is the original name and how to call that item. Be sure to write it, or paste it somewhere else so that we can use it later. In the case of a cloned teddy bear the co-string is "Sims3.Gameplay.Objects.Miscellaneous.StuffedAnimal".

37
Starting Your Code

Open up your project that you made in step "Setting up MVSC#". We are going to start off by changing the namespace and the class. The namespace is what contains classes and separates out two of the same classes. Lets name our research after Yourname Followed by mod. For Example:

Code:
```csharp
namespace KolipokiMod
```

Next we need to change the class name. Its needs to be something that will tell you what you are doing. Since we are adding a simple interaction i'm going to make it:

Code:
```csharp
public class TalkingTeddy : StuffedAnimal
```

Now you may notice in my class name that i have what functions as a deriver. What it does is it takes the code with the StuffedAnimal and assigns the TalkingTeddy to it. The deriver should be what was in your co-string. You may be wondering what your co-string was way longer then that and it was. MVSC# will the things before it for you. If it does not right click on it and hit resolve and then it should work. What it is doing is that on the top of the code it is adding a Using qualifier witch makes it so you dont have to say that name space over and over again when calling something within it.

Adding the interaction

Through this step we will be using the reflector for a while so make sure you have it opened and that you opened the dlls that we extracted in the first step.

If you do not know how the game calls an interaction it is best to see how to they do it. Because all we are doing is telling the interaction to tell us hi we need an intermediate interaction. In the Sims 3 intermediate interactions have those orange circle things next to their name. We are going to be exploring the Stereo TurnOnOff function.

Expand your Sims3GameplayObjects - Sims3GameplayObjects.dll - Sims3.Gameplay.Objects.Electronics - Stero - TurnOnOff. We now need to open the Dissassembler. To open the disassembler hit space. Once open click on the TurnOnOff to see what it does. Be sure to hit the expand method at the bottom of the dissassembler so you can see the hole code. Look through the disassembler and see how the code works.

When you have finished exploring we are going to need most of the code of the TurnOnOff Method so select the code in the disassembler and copy it. Paste it within your class definition in MVSC#.

Because this is for the stero we will need to be changing it.

Change the private sealed class name of TurnOnOff to what you want the name of it to be, make it something that describes the interaction. I'll be naming mine TalktoMe. Also make note that that class is derived from an immediateInteraction class. be sure to change the Stero to the name of your object (the first class). In my case it will say TalkingTeddy instead of stero.

Allot of this stuff we took we will not be needing and we will have to change. With your C#
knowledge you can figure out what you have to change and edit but for our example after removing
the code. You will also need to change the code so that where it says stereo you refer to your object
and where it says TurnOnOff you refer to your interaction name.

Code:

```csharp
private sealed class TalktoMe : ImmediateInteraction<Sim, TalkingTeddy>
{
    // Fields
    public static readonly InteractionDefinition Singleton = new Definition();
    private const string sLocalizationKey =
        "Gameplay/Objects/Miscellaneous/TalkingTeddy/TalktoMe";

    // Methods
    private static string LocalizeString(string name,
        params object[] parameters)
    {
        return Localization.LocalizeString(
            "Gameplay/Objects/Miscellaneous/TalkingTeddy/TalktoMe:" + name, parameters);
    }
    protected override bool Run()
    {
        //do interaction here
        return true;
    }
    // Nested Types
    private sealed class Definition : ImmediateInteractionDefinition<Sim,
        TalkingTeddy, TalkingTeddy.TalktoMe>
    {
        // Methods
        protected override string GetInteractionName(Sim a,
            TalkingTeddy target, InteractionObjectPair interaction)
        {
            return TalkingTeddy.TalktoMe.LocalizeString("TalktoMe",
                new object[0]);
        }
        protected override bool Test(Sim a, TalkingTeddy target,
            bool isAutonomous,
            ref GreyedOutTooltipCallback greyedOutTooltipCallback)
        {
            return !isAutonomous;
        }
    }
}
```

Making the Interaction Do Something
With the right code, interactions could do anything. They could start a fire to a whole lot like in the
ChaosMagePainting. They could lock doors such as in the Lockable door.

For this tutorial i will simply make it say "Hi". The Sims 3 has a nice way of setting up a
notification. It's simply:

Code:

```csharp
base.Actor.ShowTNSIfSelectable("Hello",
```
The first line in this code is calling the ShowTNSIfSelectable, which is a nice method of telling things through notifications. If you search in the reflector for the ShowTNSIfSelectable you will find that the parameters are 1) what you want it to say, 2) the theme of the notification is this case kTip (there are i believe 5 different themes, so go look them up and choose one you like), 3) thumbnail 1 and 4) thumbnail 2. This code will going into the "protected override bool Run" inside your interaction class.

Congratulations now your code will add the interaction and tell you hello. Here is what the final product should look like:

Code:
```csharp
using System;
using System.Collections.Generic;
using System.Text;
using Sims3.Gameplay.Interactions;
using Sims3.SimIFace;
using Sims3.UI;
using Sims3.Gameplay.Objects;

namespace KolipokiMod
{
    public class TalkingTeddy : StuffedAnimal
    {
        protected Sim mRevealingSim;

        public override void OnStartup()
        {
            base.OnStartup();
            base.AddInteraction(TalktoMe.Singleton);
        }

        private sealed class TalktoMe : ImmediateInteraction<Sim, TalkingTeddy>
        {
            // Fields
            public static readonly InteractionDefinition Singleton = new Definition();
            private const string sLocalizationKey = "Gameplay/Objects/Miscellaneous/TalkingTeddy/TalktoMe";

            // Methods
            private static string LocalizeString(string name, params object[] parameters)
            {
                return Localization.LocalizeString("Gameplay/Objects/Miscellaneous/TalkingTeddy/TalktoMe:" + name, parameters);
            }

            protected override bool Run()
            {
                //Do the interaction here.
            }
        }
    }
}
```
base.Target.mRevealingSim = base.Actor;
return true;

// Nested Types
private sealed class Definition :
    ImmediateInteractionDefinition<Sim, TalkingTeddy, TalkingTeddy.TalktoMe>
{
    // Methods
    protected override string GetInteractionName(Sim a,
        TalkingTeddy target, InteractionObjectPair interaction)
    {
        return TalkingTeddy.TalktoMe.LocalizeString("TalktoMe",
            new object[0]);
    }
    protected override bool Test(Sim a, TalkingTeddy target,
        bool isAutonomous,
        ref GreyedOutTooltipCallback greyedOutTooltipCallback)
    {
        return !isAutonomous;
    }
}

Now that your script is complete save the project and hit "f6". This will build the project see if it has any errors and if it doesn't it will create a nice dll for you. This dll is stored in the Visual Studio 2008 - Projects - [name of saved project here] - [name of saved project here] - obj - release - [nameofproject].dll. That file will be used to add your script to your nice cloned object and make it work. If you ever change the code be sure to run "f6" again or you will get the old dll.

Adding your script to your package
The next step in this process is to add the script to the package. First your going to need to open S3pe and then open your cloned object.

We are going to be making a new resource that will have the tag S3SA. Before we go making a new resource we need to FNV Hash and get an instance name. In s3pe hit ctrl - f or go Tools - FNV Hash. Inside the box that says text to has write the class name of your object. For me i would write "TalkingTeddy". Copy the FNV64 number. That will become the new instance.

Close out of the FNV Hash and hit Resource - Add. The Type number for all S3SA files is 0x073FAA07, and the group number is 0x00000000. The instance number is the number you just obtained in the FNV Hash. Ep flags are used to tell the object if it requires the EP or not (i assume). For those who do not have WA the EP name is 0x00. Hit ok and you will have a new S3SA type resource.
After your new resource is created go to the OBJK, Collection of Keys, 2nd Key and type "[Your Namespace Name].[Your Class Name] For this example it would say KolipokiMod.TalkingTeddy. Next you need to import your dll. Click on the S3SA and hit grid. The first item on the list should have an import/export click on it, bring the drop down menu down and click import. Navigate to your dll of your code "Visual Studio 2008 - Projects - [name of saved project here] - [name of saved project here] - obj - release - [nameofproject].dl" and hit open, after that hit commit and save your project. If all went well then you know have a nice object that will tell you Hi.

**Finishing Up**
If you testing your object in game after finishing the last step you may have noticed that the interaction would say Something like KolipokiMod.TalkingTeddy.TalktoMe . If you would like to change that open your cloned object in s3pe. FNV Hash the name that shows up in game and copy the instance in the FNV64. Find the file with an STBL file type (make sure its your language) and hit editor. A STBL Resource Editor will appear. Change the instance in the bottom to the FNV64 Hash and hit add. A new string will appear. Click on it and in the box to the right write out what you want it to say.

**Congratulations**
You have now finished your object and should have an understanding of how to make your own object mod. I cannot wait to see what you guys create.

**Special Thanks**
Special Thanks goes specially to Wito who helped me start out in object modding. I'd also like to thank Rick, Tiger, and ChaosMage for there help answering questions that i had come across. Thanks for Sri for telling me to try Object Modding.