Abstract

In this master's thesis we discuss how to solve or reduce the problems that occur when synchronizing clients in a networked environment. Irregular latency and jitter make it hard to predict where a game object will be in the nearest future. The problem is interesting because even though the bandwidth to the internet has increased for people all over the world, we still have latency and jitter in the network. Networked multiplayer games have grown in popularity, and from being played only against neighbours in a directly connected network we have started to play over the internet with our friends within the same geographical location. But there is still a problem when players want to play across continents. Too big latency and jitter will make existing games unplayable. A solution to these kinds of problems would benefit both the game developers and all the players that are playing against each other.

The main purpose of this thesis is to find out how these kinds of problems could be reduced, or in the best case be solved. The overarching theme of the solution is prediction, where prediction is based on two parts. The first part of it is extrapolation, i.e. how to make a good guess where an object will be located in the future. Extrapolation methods that are discussed are basic linear and quadratic expressions, based on the laws of Newtonian physics, Richardson extrapolation and Catmull-Rom splines. The other part of the prediction problem is to decide how an object should move from one place to another. Linear paths make objects go in straight paths between positions and this could look strange in some applications. By using some areas from mathematics behind the splines (like cubic splines, hermites and Catmull-Rom splines) we can make the movement smoother. This removes problems like objects suddenly teleporting from a place to another. To get a generic extrapolation algorithm we have discretizised mathematical expressions to get FIR-filter coefficients. With the FIR-filter coefficients, we can change extrapolation algorithm by just changing the coefficients that the algorithm will use.

To be able to test the ideas and algorithms, we have developed a game called shuffle puck and a simulation utility called Cursorsimulation. In these tools we can tune parameters and simulate a networked environment. Our work has showed that the best method to use in terms of a small deviation error is to use linear extrapolation and linear interpolation. The smoothness that we will get by using other methods like Catmull-Rom is expensive because it will give a greater deviation error. We have evolved a concept of predictors and replicators that are responsible for how a game object will move in an asynchronous networked environment. We have shown that this will work, but will not be as good as the existing methods as client side prediction, that is common in modern games.

Table of contents

1 Introduction

1.1 Background

Today's market of games has a wide variety of genres running on different platforms. We can see games on consoles, computers and cellular phone etc. But since the middle of the 1990's when internet grew in popularity and accessibility the game developers have tend to focus more on making a multiplayer part of their games. In this thesis we will look at the problems programmers face when it comes to developing multiplayer games that use a networked environment. The primary discussions are about extrapolation and interpolation. The way we are going to solve extrapolation and interpolation is by going from standard linear methods to cubic polynomials, where *splines* are one area. We are going to use ideas from the control theory like *Kalman* filtering and FIR (*Finite Impulse Response*) filters. The methods discussed in this thesis could also be applied to other topics that involve controlling an object, where there is a delay between sending a command and until the receiver reacts to that signal.

A *shuffle puck* game has been implemented in *Python* with the use of *Twisted*, a network API for writing asynchronous applications in Python. Using an asynchronous approach gives rise to some new problems that we don't face when it comes to a synchronized approach. The main problem is that we get non-uniform updates and many extrapolation and interpolation algorithms require that updates are in a uniform matter. We briefly discuss some of the concepts that Twisted use. To be able to get results we have also developed a utility called *Cursorsimulation*, which can simulate a network and where we can make movement paths. Cursorsimulation will use these paths in conjunction with extrapolation and interpolation algorithms to produce a result.

1.2 The problem

Today, people around the world can and do play computer games over an internet connection with each other. This fact makes a game programmer have to consider new problems that did not exist at all when it comes to single player game programming. Problems like how players exchange data with each other, securing for cheats, synchronizing the game states in the game, etc. In this thesis we are going to focus on how to synchronize game players that are playing in the same virtual world, i.e. we will try to make it possible to show a player's movement and actions instantaneous on the other participants' computer screens. The main reason for the synchronization problems in a networked environment is that the packets sent between participants have a travel time. Another problem is that there is no guarantee that packets arrive in a given amount of time: the packets arrive in a non-uniform way.

This is an interesting problem both for game developers and for the actual players that spend a lot of time playing networked computer games. Networked games have grown from just being a game that we are playing with a neighbour to play on the internet with friends that are in the same geographical area (same country). However, if we try to play games with players on other continents, we will get horrible results using the

existing algorithms used in computer games. The high latency that arises between continents makes it almost impossible to have an internet tournament with players from, e.g. Sweden, Australia, China and USA. An investigation of how and if it is possible to solve these kind of problems would have made all participants in the gaming community happy.

What could then happen if we don not take care of the problems with the delays? The delay may be acceptable in many games, but in realtime games as first person shooters, the delay can have a devastating effect on the games' playability. Consider two players playing against each other. The game takes place on a server that both players connect to. One player has an average packet delay of *50 ms*, and the other player has about *200 ms* average delay of the packets. The decisions of the game are made on the server, i.e. it is the server that has the true game state. If the player with the lower packet delay fires a bullet towards the other player and hits him, then he will get a response from the server that he hit him much faster than the other player will. When the player with the higher packet delay receives the message from server he could already be out of sight from the other players' point of view, but get hit anyway, i.e. he will get a feeling that he has been shot from behind a wall.

1.3 Terminology

This is an explanation of the terms used in this report.
Entity: An object in the modelled world.
State: A snapshot of all entities and their variables that exist in the modelled world.
S**tate update**: An operation that updates the variables and object states in the game.
Message: A message for communication between the server and clients.

1.4 Outline

This report is structured so that we in chapter two start with a theoretical part where we get the knowledge of why and how extrapolation and interpolation are used. We also get experience with splines, Kalman filtering, Richardson extrapolation and digital filters. With this knowledge we are able to discuss and evaluate different algorithms and approaches. This is the main topic for the last part of chapter two. In chapter three we describe the general structure and concepts of a networked multiplayer game. We start the chapter by introducing Twisted, which as mentioned earlier, is an asynchronous framework for networked applications written in Python, which has been used in the test application, called shuffle puck that we have developed for this thesis. Chapter four is the last chapter in which we summarize the evaluation of the project. Here we find a section where we can read how shuffle puck is implemented and why we have used the chosen approach. To get true results from the different interpolation and extrapolation methods there is a section about a utility called Cursorsimulation that has been developed for this thesis. Furthermore we will find a section about how some of today's games try to solve the latency and jitter problem. Last in chapter four we will find the conclusions of this project and how this thesis could be continued.

2 Prediction

In this chapter we are going to look at the theory that we are using to be able to minimize the effects of the latency in the network. Most of the sections are related to extrapolation and interpolation. So after a general description of how prediction works and why we should use prediction in computer games we will discuss the basic concept of interpolation and extrapolation. After that we will look at more specific algorithms that we could use. One section is about Kalman filtering where we have an example that shows how Kalman could be used for our problem. We do some calculations to get acquaintance of how the methods will perform and look when they are plotted. Last in this chapter we will derive FIR-filter coefficients that are used in a generic extrapolation algorithm.

2.1 Introduction to prediction

Prediction is a concept that allows us to guess how things are going to be in the future. It is used in many areas: some implementations rely on complicated mathematics; some are based on human guesses and quite often combinations of both. Prediction is, from a mathematical view, a concept for finding a set of points in the future based on the knowledge of the points in the past. The quality of the prediction often depends on how much information you have about the past. If we for example should predict the path of a car we would probably get better result if we knew the cars velocity, acceleration and at what time it was at the last position, than if we only knew it's position.

2.1.1 Prediction in computer games

Why do we need to do prediction in computer games? Well, in almost every game we have a game loop where we are doing similar things in each frame. The most common loop is:

- Receive messages from other players
- Do game logic calculations, like, handling input, collision detection etc.
- Send an update to the server (and to other players) with your new state
- Render the graphics onto the screen

Within each frame it is very likely that there are missing or delayed messages from other players. If we just ignore that the messages were missing or delayed we would get jerky movements of the other players' entities in the game. What prediction does in these cases is that it tries to make a good guess where the players were supposed to be in the particular frame. The result heavily depends on what kind of prediction algorithm we are using and what kind of movement the player was doing. We should be aware that prediction does not make jerkiness to go away; it will most probably just make it better. To be able to get rid of jerkiness we will have to use a smoothing algorithm, i.e. interpolate between the predicted points.

2.1.2 Movement patterns make sense

Objects that are easy to predict, and with good result, are objects that are moving in a linear way, like an airplane in a flight simulator. It's not likely that an airplane will do an instant 180 degree turn. It's more likely that it will travel in the air with approximately the same speed and only do small angular corrections. On the other hand, in games like for example football games or 3D-shooters the players tend to do more irregular movements. Here is it very possible that a player is running forward, and then suddenly changes direction. The uncertainty and non-uniform movements makes prediction very hard to do.

2.2 Interpolation and extrapolation methods

2.2.1 Extrapolation

In the previous section we talked about the concept of prediction. Extrapolation is the mathematical way to express the part of the prediction where we guess the new position for an object. When doing extrapolation, time is often involved. For example an, object on the computer screen has in 2D one y- and one x-coordinate and when we extrapolate a new position we have to scale it by the amount of elapsed time that has passed. Extrapolation is always easier to do when the time step is uniform. When that is not the case we have to use dividing differences to get the correct result.

2.2.2 Dead reckoning

The concept of the dead reckoning algorithm is that everyone participating in a virtual world is using the same algorithm to extrapolate, i.e. predict where the entities in the world will be in the future. From the beginning the dead reckoning evolved from the United States military, they had need for doing simulation on their vehicles. And for this they developed the Distributed Interactive Simulation (DIS) protocol [1].

In dead reckoning an entity not controlled by the player is known as a ghost. The entity that represents the player is controlled by the player itself. The player uses dead reckoning on it own entities to be able to decide when the extrapolation has too much deviation from a specified threshold. A message that contains information about position, velocity and acceleration is sent to the other players when a threshold is exceeded, so they can restart the extrapolation with new values. The deviation that the player calculates is based on the information of the last message he sent to the other players and the true state of the entity. In this way both the owner of the entity and the other players are going to extrapolate the entities in the same way. Because of that dead reckoning concept uses extrapolation it also suffers from jerkiness. The propagation of the error depends on the entities' velocity and on the rate that messages with state information arrives. Accuracy is gained by sending more messages, but with more messages the bandwidth usage will increase.

2.2.3 Interpolation

Interpolation is the way of how to calculate what the value at the points in between the known points should be. There are several ways to do this. It can be achieved by going from easily calculated linear expressions to more advanced higher order polynomials.

C^0 continuity

Continuity in between points is achieved by going from one point to the next point. But depending on whether the *n*th derivative on the start point and the end point in each curve segment is the same, we get different kinds of smoothness conditions. Smoothness condition is defined as a class C^k. A curve $s(t)$ is said to belong to the class $C^k[a, b]$ if it has k derivatives $s'(t)$, $s''(t)$, …, $s^k(t)$ and all of them are continuous [2]. When we have C^0 continuity we just connect our interpolated segments and we do not have any continuous derivative.

C^1 continuity

By the same definition of C^k as in previous section we get our first smoothness condition if we have a continuous first derivative between the end point in the adjacent segment and the start point in the next segment. This is a condition we need to enforce to be able to get a smooth curve between interpolation points. This is illustrated in the figure below. We can see that $P_3 = Q_0$ (C^0 continuity) and that $P_3' = Q_0'$. Interpolating polynomials always have this condition fulfilled, but because we only want to use polynomials of lower degrees, we have to use a chain of different polynomials in between the predicted points. This leads us to *splines* that we will discuss in section 2.2.5.

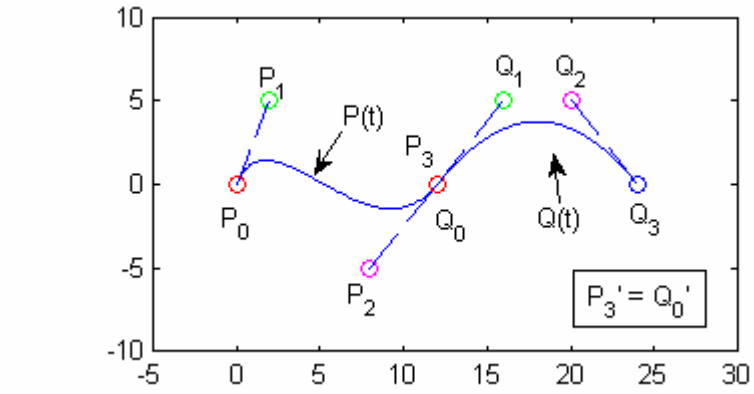

Figure 1: *Illustration of C^1 smoothness condition*

2.2.4 Interpolation with polynomials

When we only know a few points of a curve we can try to find a polynomial that interpolates our known points. By going for a higher degree of the polynomial we can get more natural results than by using straight linear paths. The most common way to find a polynomial is to use *Lagrange's-* or *Newtons'* interpolation formula. Newton's interpolation formula is more effective than the Lagrange version mainly because of that if you want to calculate a higher order polynomial you have to redo all the calculations from the beginning with Lagranges' method, whereas with Newton's

method you can use the previously calculated values. We will use Newton's method. When calculating the polynomial we state that our expression should be of this form

$$P(x) = C_0 + C_1(x - x_1) + C_2(x - x_1)(x - x_2) \qquad \textit{(eq. 2.1)}$$

To get our polynomial we have to solve the equation for the C_i coefficients (see appendix). When it is done we have a polynomial that we can use for any point in between the interpolation points. We can of course also use the polynomial to calculate points that lay outside the known end points. One might think that adding more degrees to the polynomial will gain better prediction. That is not the case. Higher order polynomials often lead to very big fluctuations between the interpolated points, the most famous example which is called *Runges phenomena* [3]. This is the reason why we stick to second order interpolation polynomials for prediction.

2.2.5 Splines

The main idea with splines is to use different polynomials to approximate a curve or a function in an interval. Assume that we have a curve in an interval *[a, b]* and that we have divided the interval with joint points $a \leq x_1 < x_2 < \ldots < x_n \leq b$. Then the easiest way to approximate the curve is to draw straight lines between the points in the interval, i.e. the same method as we proposed in the section on interpolation with polynomial. This is illustrated in the figure below where a sinus curve is represented by straight lines between the points.

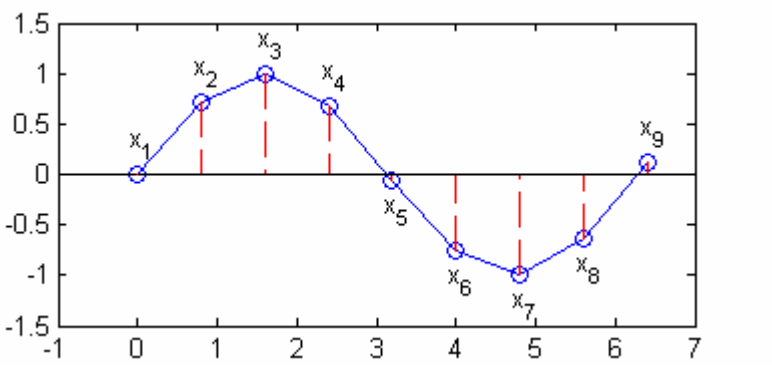

Figure 2: *Spline with linear path in between segments*

If we instead use a polynomial of higher degree we will get curves between the joint points. And it's also possible to get C^1 continuity at the joint points. Doing this with a polynomial with degree three is called a cubic B-spline or just simply cubic spline. When we are computing cubic splines we are working with four points, a start point P_0, an end point P_3 and two points in between, P_1 and P_2. The two points in between are also called *control points* or *influence points* and in computer games they can represent the velocity of an entity. The figure below illustrates a typical cubic spline.

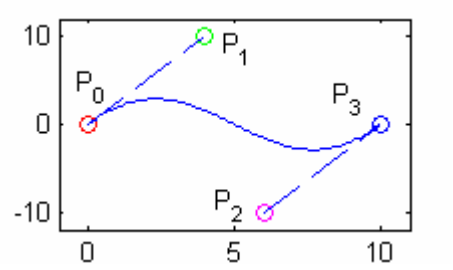

Figure 3: *Illustration of a cubic spline and its control points*

Catmull-Rom splines

A Catmull-Rom spline is a spline that interpolates all the points we give to it. These types of splines are perfectly smooth, i.e. C^1 continuity is fulfilled at all its points. The derivatives at the points that are not end points are easy to calculate. The derivatives in these points are parallel to a line drawn between its adjacent points. In the figure below we can se that the tangent at P_1 is parallel to the vector $P_0 P_2$.

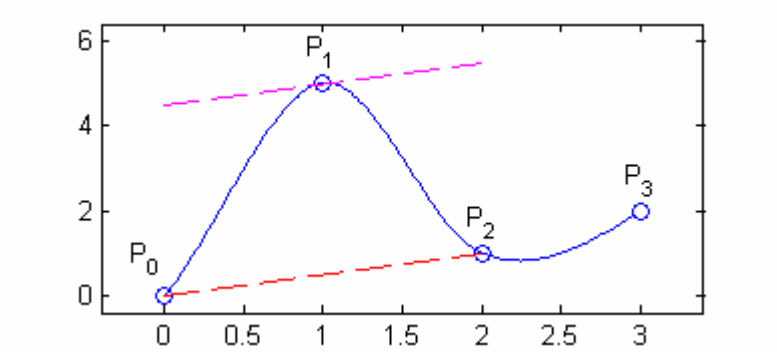

Figure 4**:** *Visually showing Catmull-Rom tangents calculations*

How are the tangents at the end points in the picture above calculated? They are calculated using an end formula that is

$$P_i' = \frac{1}{2}\left(3P_i - 4P_{i+1} + P_{i+2}\right) \qquad \textit{(eq. 2.2)}$$

This formula works for both the start point and the end point of the curve.

2.2.6 Kalman filter

In many processes we can not do measurements exactly. Measurements could be limited by the tools we are using; it could also be that there is disturbance from our environment that affects the process. In 1960 Rudolf E. Kalman and Richard Bucy developed an algorithm for control theory [4]. This algorithm, which goes under the name *Kalman filter*, makes use of imprecise data in a linear system that is affected by Gaussian noise in the process. With that imprecise data a Kalman filter can estimate the state variables of a process. It can be shown that of all existing filters, it is the Kalman filter that minimizes the variance of the estimation error [5]. Kalman is widely used in navigation systems, radar tracking, sonar ranging and satellite orbiting control to give a few examples.

To be able to use the Kalman filter we need to establish a model of the process we are interested in. This model is described by a linear system (if it is not linear we have to linearize it). A linear system can be described by two equations: a state equation and an output equation. The state equation tells us about the dynamics of a system, and the output equation gives us a measurement of the states. Often we only measure one of the states. We get this linear system

$$x_{t+1} = Ax_t + Bu_t + w_t \qquad \textit{(eq. 2.3)}$$

$$y_t = Cx_t + z_t \qquad \textit{(eq. 2.4)}$$

In the state equation A is a state transition matrix, x is a state vector of the states that we are going to estimate, B is a matrix for the input to the process, u is the input and w is the noise of the process. The noise has to be Gaussian noise with zero mean. What this means is that the noise should not drift in any direction. This kind of noise is also known as *white noise*.

When we have decided what state variables to use and have setup a model we could start using the Kalman filter and calculate an estimate of x (we denote estimated values by the hat sign, so $\hat{x}$ is the estimated vector). To compute the Kalman filter we have to calculate five equations. Two of them are updating equations. They are responsible for projecting the current state and error covariance estimates forward in time to be able to obtain the estimates for the next time step. These estimates are also called *a priori* estimates. Basically what they are doing is prediction of the new states. The other three equations are responsible for feedback. A feedback that transforms the *a priori* estimate into an improved *a posteriori* estimate, i.e. these equations is the *corrector* equations. We have the following equations [6]

$$\bar{x}_t = A\hat{x}_t + Bu_t \qquad \textit{(eq. 2.5)}$$

$$\bar{P}_t = AP_tA^T + Q \qquad \textit{(eq. 2.6)}$$

$$K_t = \bar{P}_tC^T\left(C\bar{P}_tC^T + R\right)^{-1} \qquad \textit{(eq. 2.7)}$$

$$\hat{x}_{t+1} = \bar{x}_t + K_t\left(y - C\bar{x}_t\right) \qquad \textit{(eq. 2.8)}$$

$$P_{t+1} = \left(I - K_tC\right)\bar{P}_t \qquad \textit{(eq. 2.9)}$$

Here we used the bar as notation that it is extrapolation from an earlier state. This notation should in fact have index $t+1$, but because it's only used as a temporary variable we denote the index with t only. We have introduced Q and R into the model. Q is a covariance matrix for the process noise and R is a covariance matrix for the measured noise. They are defined as

$$Q = E\left(w_t \cdot w_t^T\right) \qquad \textit{(eq. 2.10)}$$

$$R = E\left(z_t \cdot z_t^T\right) \qquad \textit{(eq. 2.11)}$$

P in the equations is a covariance matrix of the error in estimating x. The Kalman gain is denoted by K. We can see that if the measurement noise is large, Q will be large and then the Kalman gain will be small and it will not contribute with a large value to the estimation of the estimate of the next $\hat{x}$. Now we have described the Kalman filter. To use a Kalman filter we initialize the variables and then repeat the Kalman filter equations in every time step.

2.2.7 Richardson extrapolation

Finding derivatives can be done in several ways. Richardson extrapolation is a method that gives a better estimate of the derivative than we would have obtained using only dividing differences to calculate the derivatives. Assume that the value of a function *F(h)* can be calculated for different values where $h \neq 0$ and we search for the limit of the function when $h \rightarrow 0$. For central differences this can be calculated as

$$F(h) = \frac{f(x+h) - f(x-h)}{2h} \qquad \textit{(eq. 2.12)}$$

Where h is any value not equal to *0*. Our problem is now to find the derivative of the function

$$f'(x) = \lim_{h \to 0} F(h) \qquad \textit{(eq. 2.13)}$$

which written as Richardson extrapolation will be

$$f'(x) = F(h) + \frac{F(h) - F(2h)}{3} + \mathrm{O}(h^4) \qquad \textit{(eq. 2.14)}$$

We now have an estimate of the derivative with a truncation error $O(h^4)$. Without Richardson extrapolation we would have got a truncation error of $O(h^2)$ instead. We got the expression by using *Taylor series*. See the appendix for calculations more in detail.

2.2.8 Digital filters

Digital filters are filters that work with discrete time intervals. The idea with digital filtering is to use some constants that are multiplied with the signal and added in each time step of the filtering process. The results of the calculations are the output from the filter. Depending on how the designers of the filter choose the constants we get different behaviour of the filter. The most common way to design filters is to make them cut off frequencies that are not desired, where low pass, high pass and bandwidth filter being the most well known ones.

FIR filter

FIR means *Finite Impulse Response*, i.e. the input to the filter is finite. The reason why it is finite is because of that there is not any feedback in a FIR filter. So if you put an impulse as input signal (a single input value of one followed by many zeros), the output will return to zero as soon as the single non-zero value has passed through the filter. The filters is made of multipliers, adders and delay elements (see figure 5 below that shows

a FIR filter in a block diagram). In a FIR filter the word *tap* is often mentioned. A tap is a pair of a coefficient and a delay. The order *N* of a FIR filter is the number of taps used in the filter. The higher order of a filter the more memory it will use. And it will do more calculations, but higher order of the filter also gives the filter designer more possibilities to achieve the desired result. Thus there is a trade off between simplicity and effectiveness.

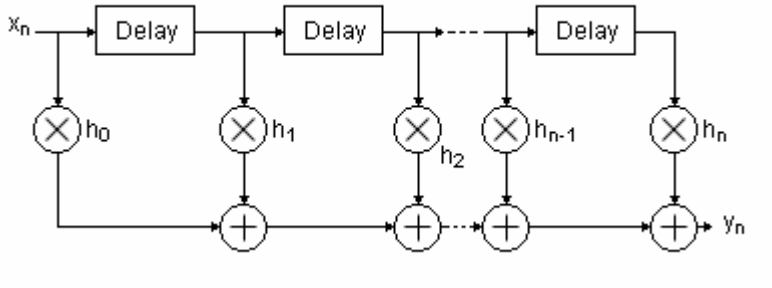

Figure 5: *Illustration of a FIR-filter*

So the output *y(u)* of the filter is

$$y_n = h_0 \cdot x_n + h_1 \cdot x_{n-1} + \ldots + h_n \cdot x_0 = \sum_{i=0}^{n} h_i \cdot x_{n-i} \qquad \textit{(eq. 2.15)}$$

Basically what a FIR filter does is producing a weighted average of its *N* most recent input samples [7].

IIR filter

An IIR filter works in a similar way as a FIR filter. The big difference is that it uses feedback from the output signal to the input in the next computing step. So when the filter is subjected to a signal, then its output need not necessary become zero when the entire signal is through the summation of the filter. The response of a signal in an IIR filter is infinite. Because of the recursion we have to be careful when designing an IIR filter. An incorrectly built filter can make the filter unstable and cause the output signal to escalate.

2.3 Prediction applied

Now that we have discussed the theory that we are using in our prediction methods, we are going to actually use the theory and the ideas to evaluate how we could use them in practice. In the following sections we will discuss the drawbacks and the benefits using the prediction methods in a computer game.

2.3.1 Evaluation of prediction schemas

When we evaluate how a method works in practice, we are going to use two functions, sinus and logarithmic, that simulates the true positions. These functions are a good choice for simulation. The sinus wave with that few points (one every natural number), makes prediction hard. The logarithmic function is quite smooth and easy to predict. To be able to have some numbers to look at we have in each plot calculated the error of a mean value as shown for sinus below

$$mv_{err} = \frac{1}{(n \cdot 0.01)} \cdot \sum_{i=0.01}^{n \cdot 0.01} \left| \sin(x_i) - iv(x_i) \right|, \quad n = 3\pi \text{ , } iv = \text{interpolated value}$$

(eq. 2.16)

In the plots we notice that we do not start our predicted path until we have a few true positions. This is due to that we can not calculate any extrapolation without any true positions. When implementing a computer program, we could just initialize the predicted start point to the first position. This would give a strange start of the object, but it is just a matter of very little time before it is running as it should.

Basic prediction

The most basic form of prediction is the point to point method. This method renders the object in the same position until a new state message arrives. When a new message has arrived, the entity is moved to the new position that the message states and will now render the entity at that position until the next state message arrives. This method requires a lot of messages if it is to be able to have a fairly good result. Because of that a server sends updates at a specified frequency. Which is much lower than the players' framerate. This method is not used very often in computer games with a lot of moving objects. Due to that this method doesn't use any kind of prediction it will always render a player too late in terms of a synchronized world. The message has a time Δt from that it was generated to that the player receives it, and the player will always be rendered too late with that amount.

A better idea is to use a linear prediction method. From Newtonian physics we know that $s(t) = v \cdot t$. By using this and the last known position we can extrapolate a new position. We have this linear expression

$$p(t) = p_i + v_i \cdot t \qquad \textit{(eq. 2.17)}$$

The major drawback with this method is that it assumes that the object has constant velocity. If the actual velocity is not constant, then we are going to get a deviation from the actual position and the predicted position. The magnitude of the deviation depends not only on the objects velocity. It also depends on Δt. Using this method leads to objects that moves in straight lines and suddenly changes position, direction and velocity. Using this method which obviously gives us C^0 continuity gives the following plots and value's.

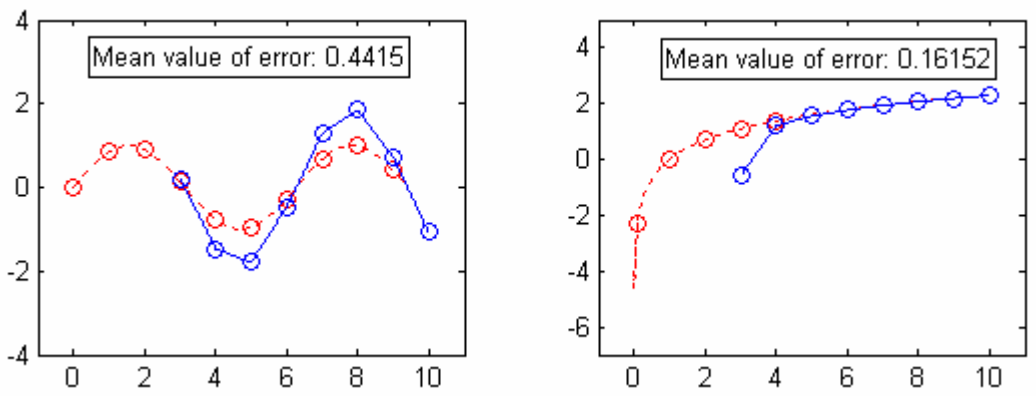

Figure 6: *Quadratic prediction and linear interpolation*

If the problem with the linear prediction were that we had constant velocity as a requirement for making good prediction, then we could add one more term, the acceleration, to the equation and get a higher order polynomial. Doing so gives us following equation (see appendix for derivation).

$$p(t) = p_i + v_i \Delta t + \frac{1}{2} a_i (\Delta t)^2 \qquad \textit{(eq. 2.18)}$$

Will this work? Well, the short answer is sometimes. Again it depends on how big Δt is. Small intervals between real state updates make this method perform slightly better than using only linear prediction. But when Δt grows, we can get horrific results due to that the acceleration is multiplied with the squared time and the end result is even worse than using only linear prediction.

Catmull-Rom prediction

As we described earlier in this chapter, the Catmull-Rom spline gains perfectly smooth curves, i.e. C^1 continuity is achieved, but this is only the case when we already know the points that we interpolate between. Consider that we have a set of points and we calculate a Catmull-Rom spline. The derivative in the last point will be calculated with the end point formula. We know that the derivative in the last point only depends on the last three points. But if we now add one more point and recalculate our Catmull-Rom spline we won't get the same derivative in the last but one point. This is because our tangent is now calculated as the normal tangents in a Catmull-Rom spline. I.e. using the approach with continuous use of end point formula makes our spline end up with C^0 continuity.

A common way to draw smooth Catmull-Rom curves is to only draw the segment in between the two middle points. To use this approach in a computer game, the game must have regular updates with very small delay in between the position updates. Otherwise the players are going to notice the delay that they are exposed to. In the picture below we can see how it would look like when following a sinus curve by using Catmull-Rom interpolation between quadratic predicted positions. As we expected with this approach we can see that there isn't any interpolation done between the last two predicted points.

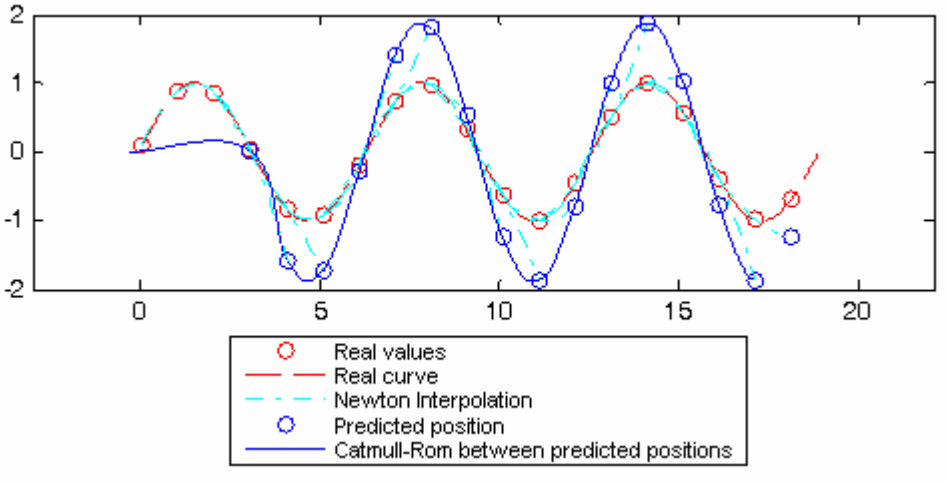

Figure 7: *Catmull-Rom interpolation between points*

Can we use Catmull-Rom to extrapolate the positions then? Yes, if we enforce that the derivative should be the same in the second last point as it was when it was the last point itself. This approach is described in detail in the coming section about extrapolation with digital filters.

There is a mathematical concept that is called *hermite geometry* [8]. It is basically the same as a Catmull-Rom spline, but it is calculated using a start point and an end point. And instead of using points in between for specifying the curves' shape, the hermite geometry uses the derivatives at the start and the end points for this. Knowing that, we can use same derivatives the end point and the start point in the next segment of the spline.

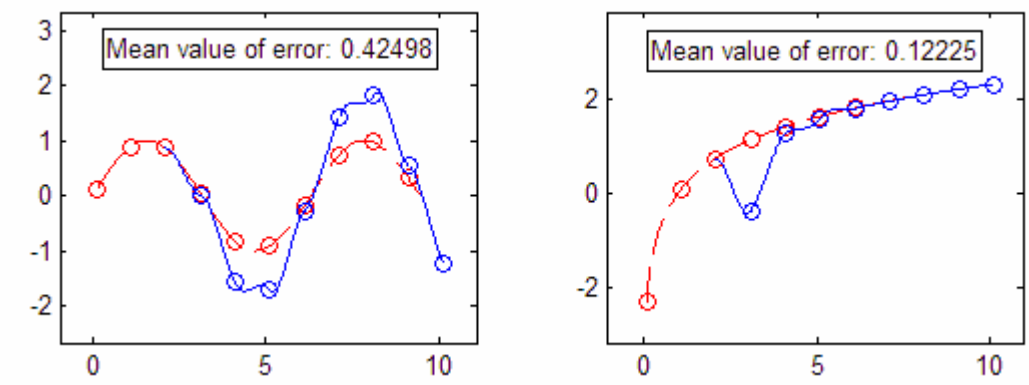

Figure 8: *Quadratic prediction, (left) and hermite interpolation*

Basically our problem is to calculate the end derivatives. We use the same formula as the end formula that we described in the section about Catmull-Rom and C^0 continuity. This method makes quite a smooth and nice interpolation between the points. The results of the sinus and logarithmic plot are seen in figure 8. The math behind the hermite geometry can be found in the appendix.

Cubic spline

As illustrated in figure 3, a natural cubic spline interpolates its start and endpoint, but not the two points in the middle. We could make use of this with a little different approach than before. The idea is based on an algorithm that Nick Caldwell, MIT, proposed in an article at GameDev.net [9].

We use the same notation of the points P_0, P_1, P_2, and P_3 as before. But now point P_1 represents the direction and velocity with which the object leaves the start point. P_2 sets the direction and final velocity that the object is supposed to have at its end point. I.e. we have tangential lines between P_0, P_1 and P_2, P_3 that the spline should follow at the start and at the end of the curve.

If we knew everything about the state in the start position and the end position, then it would have been a straight forward calculation of a cubic spline. But due to the latencies in a networked computer game, we won't know exactly what the states look like and the exact locations of the points. We have to extrapolate a new position again. However we can make use of the points that were extrapolated in our cubic spline. The following equations system yields a cubic spline which intersects with its start point and end point [10].

$$\begin{cases} x = At^3 + Bt^2 + Ct + D \\ y = Et^3 + Ft^2 + Gt + H \end{cases}, \; 0 \le t \le 1 \qquad \textit{(eq. 2.19)}$$

The equation system has to be solved for each coefficient (see appendix). We notice that it is the same equations for both dimensions, thus if we have a third dimension we would just have added another set of the same equations. How can we use this knowledge to smoothing the deviation that occurs when predicting? We have to track both the true positions in the states and the predicted positions. The predicted path is based on the two last true positions. So we are using linear extrapolation to get a new predicted position (we could of course also use any extrapolation method here, but for the sake of simplicity we use linear extrapolation). When we get a position update from the network we start a new linear path prediction. We can calculate where we expect our entity will be when we received next position from network. Right now we assume that the updates from the network arrive in uniform time steps. As start point for our cubic spline we will use the position that our entity is at the moment we get a position update. And as end point we will use the predicted position at time t_{update}, i.e. we will move along a cubic spline between predicted points, and not the true points. By doing this we will fluctuate around the true positions and the magnitude of the fluctuations depend on how good the prediction is. The advantage of doing it this way is that we will

get rid of jerkiness when we get new position updates i.e. we get C^1 continuity as we wanted.

But how do we know where the two control points in the middle should be? The first control point gets the position where our entity should be in $1/3\Delta t$ seconds (Δt is now $t_{update} - t_{now}$) if we continued our last prediction path. And the second control point should be placed in a similar way. But here we do the prediction with start from the end point and with the same, but negative, velocity from our last prediction. We place the second control point where it should have been at time $2/3\ \Delta t$.

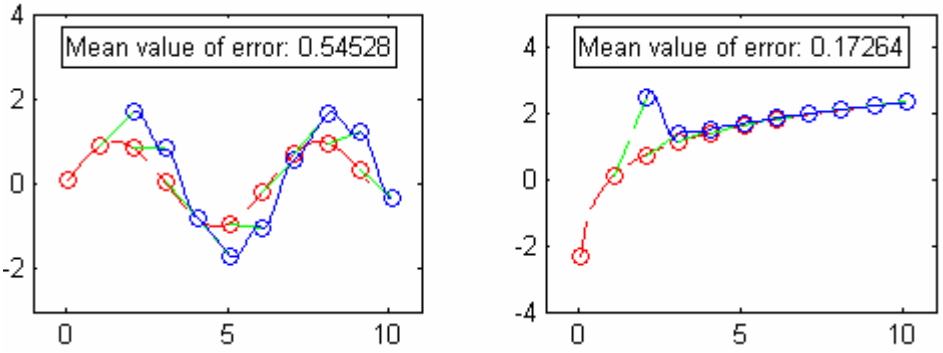

Figure 9: *Linear prediction and cubic spline interpolation*

In the above figures we can clearly see that when we have bad prediction, we get larger fluctuations and also greater curves in between predicted points. One major drawback with this method is that it will always make two turns between two predicted points, and the turns are quite noticeable when the prediction is bad. So this idea is most likely to be best suited for games where we do not have too much irregular movement.

Kalman filtering

The first thing we have to do to be able to use a Kalman filter is to decide which states we are going to use in the process model and what our input to the system is. If we choose to use a Kalman filter to follow the path of an object, we could make use of the object's actual position, velocity and acceleration. When we try to calculate the new position, we use that

$$p_{t+1} = p_t + v_t \cdot dt + \frac{1}{2} a_t \cdot dt^2 \qquad \textit{(eq. 2.20)}$$

and for the velocity we have the following relation

$$v_{t+1} = v + a \cdot dt \qquad \textit{(eq. 2.21)}$$

These two equations define the state vector x of the system. Identifying these equations in the linear system gives a new relation that in matrix form will be

$$x_{t+1} = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix} \cdot x_t + \begin{bmatrix} dt^2/2 \\ dt \end{bmatrix} \cdot u_t + w_t \qquad \textit{(eq. 2.22)}$$

The measured output should be only the position i.e. not a vector containing both position and velocity. This is because we can only measure the position of the object. The output equation will be

$$y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot x_t + z_t \qquad \textit{(eq. 2.23)}$$

Now that we have a state model of the system we could continue using the Kalman filter by computing the Kalman filter equations. Let us say that an object in a game is travelling 300 pixels/s and that we have a jitter that makes the packets in the network have an average fluctuation of about ±20 *ms*. The jitter of the latency makes the uncertainty of the measurements of the object's position be ±300·0.02 *pixels/measurement* = ±6 *pixels/measurement*. This is the measurement noise z_t in equation 2.23 above. Let us further say that the object is commanded to an acceleration that is 20 *pixels/s*. This leads to a process noise, w_t, that is ±200·0.02 *pixels/measurement* = *±0.4 pixels/measurement*.

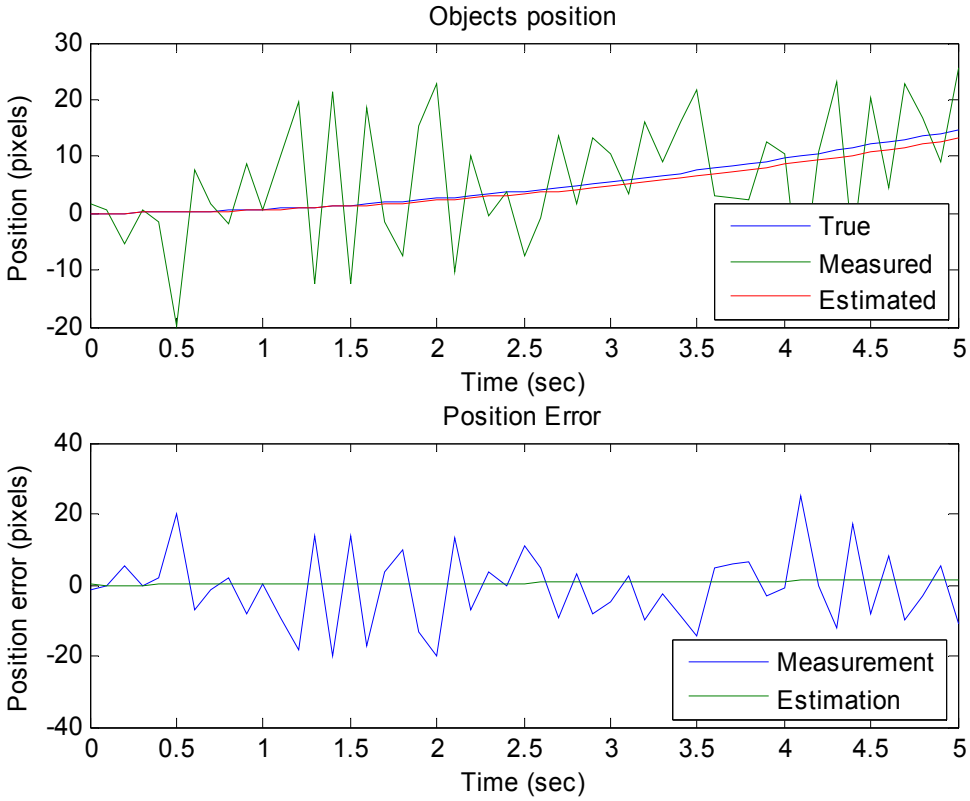

Figure 10: *Kalman filtering example of an object following a path*

In the picture above we have simulated these conditions in Matlab. But to be able to distinguish the difference between the true and the estimated position in the picture we had to lower the acceleration input. In the picture, acceleration is set to 1 pixel/s. The picture really shows the power of Kalman filtering. We clearly see that the estimated position is a lot better than the measured position is.

2.3.2 Extrapolation with digital filters

The FIR filter concept can be used to express the above algorithms in the same manner, i.e. a weighted sum of input signal and constants. If we discretisize the equations to handle only the positions we get equations in the same format as the FIR filter equation. If we start with the simplest case, the linear prediction, then we know that the next point can be calculated using the last position and velocity in this equation

$$x = x_i + v_i \Delta t \qquad \textit{(eq. 2.24)}$$

With uniform time step we get that $\Delta t = 1$. And the velocity is $(x_i - x_{i-1})$. Thus we get

$$x = x_i + (x_i - x_{i-1}) = 2x_i - x_{i-1} \qquad \textit{(eq. 2.25)}$$

However if we do not have uniform time step we have to take care of it by using dividing differences. Then we get a modified version of the above equation, like this

$$x = x_i + \frac{(t - t_i)}{(t_i - t_{i-1})}(x_i - x_{i-1}) \qquad \textit{(eq. 2.26)}$$

We can verify this by a simple example. Let us say that we have the following situation

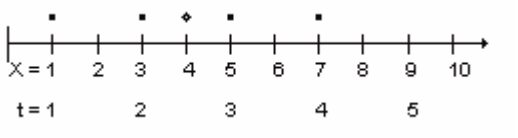

Figure 11: *Linear extrapolation example*

If the situation is that we have our last point, $x_i = 4$ and we get our next update at $t = 4$, then we know from the linear equation with uniform steps that the next points should be at $x = 7$ as shown in the figure above. We use the modified version to show that we also get the same answer

$$x = 4 + \frac{(4 - 2.5)}{(2.5 - 2)}(4 - 3) = 4 + 3 \cdot 1 = 7 \qquad \textit{(eq. 2.27)}$$

We get the expected result. Now we continue to show how to express the extrapolation methods discussed above in a similar manner.

When it comes to our prediction with velocity and acceleration we can approximate the acceleration by using the difference of the last two known velocities. I.e. $a = (v_i - v_{i-1})$, then we can write our formula in this way

$$x = x_i + (x_i - x_{i-1})\Delta t + \frac{1}{2}(v_i - v_{i-1})\Delta t^2 \qquad \textit{(eq. 2.28)}$$

With uniform time step and a and v substituted into the equation as positions gives

$$x = x_i + (x_i - x_{i-1}) + \frac{1}{2}((x_i - x_{i-1}) - (x_{i-1} - x_{i-2})) =$$

$$= \frac{5}{2}x_i - 2x_{i-1} + \frac{1}{2}x_{i-2} \qquad \textit{(eq. 2.29)}$$

For Catmull-Rom we have the end point formula for the tangent in the last point, that is

$$P_i^{'} = k = \frac{1}{2}\left(3x_i - 4x_{i-1} + x_{i+2}\right) \qquad \textit{(eq. 2.30)}$$

When we know the tangent in the last point, then we can use that to extrapolate the next point, which will be

$$x = k + x_i \quad \Leftrightarrow \quad x = \frac{5}{2}x_i - 2x_{i-1} + \frac{1}{2}x_{i-2} \qquad \textit{(eq. 2.31)}$$

In this case we do not have to consider what Δt is, because the derivative will provide us with an average step size to use.

We can do the same for Richardson extrapolation. But we would not use central differences as we did in the theory part. The simple reason is that we do not know the point in the future, i.e. the *f(x+1)*. Instead we are going to use the backward difference, which is basically the same thing as central difference. The only difference is that we use two points instead of three points. This gives a slightly more inaccurate estimate of the derivative. The expression for backward difference is

$$F(h) = \frac{f(x) - f(x-h)}{h} \qquad \textit{(eq. 2.32)}$$

If we substitute that into the expression for Richardson extrapolation and we use uniform time step, i.e. *h=1*, then we get

$$f'(x) = F(1) + \frac{F(1) - F(2\cdot 1)}{3} \quad \Rightarrow$$

$$f'(x) = f(x) - f(x-1) + \frac{\left(f(x) - f(x-1)\right) - \left(\dfrac{f(x) - f(x-2)}{2}\right)}{3} \qquad \textit{(eq. 2.33)}$$

Now that we have this expression we could use the same formula (eq. 2.31) as the one we used in Catmull-Rom above to solve where our next extrapolated point will be. After substitution and summarizing the terms we get that

$$x = \frac{7}{3}x_i - \frac{5}{3}x_{i-1} + \frac{1}{3}x_{i-2} \qquad \textit{(eq. 2.34)}$$

To be able to use this approach with a non-uniform time step we have to substitute *h* in the Richardson equation with $t_0 - t_1$ and $t_0 - t_2$ depending on which backward difference we evaluate. We get that the derivative is

$$f'(x) = \left(\frac{4}{3(t_0 - t_1)} - \frac{1}{3(t_0 - t_2)}\right)x_i - \frac{4}{3(t_0 - t_1)}x_{i-1} + \frac{1}{3(t_0 - t_2)}x_{i-2} \qquad \textit{(eq. 2.35)}$$

Solving this analogous to equation 2.31 we get that the expression for Richardson extrapolation with uniform time steps is

$$x = \left(\frac{4}{3(t_0 - t_1)} - \frac{1}{3(t_0 - t_2)}\right)(t - t_1)x_i + \left(-\frac{4(t - t_1)}{3(t_0 - t_1)} + 1\right)x_{i-1} + \frac{(t - t_1)}{3(t_0 - t_2)}x_{i-2}$$

(eq. 2.36)

We conclude that we can write many extrapolation methods in the same fashion, i.e. as a weighted sum of input signals. We can write this as a linear system in matrix form

$$\bar{x}_t = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ 0 & \ddots & 0 & \vdots \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \bar{x}_{t-1} + \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \cdot u \qquad \textit{(eq. 2.37)}$$

Extrapolation method	**Coefficients *h***, $[h_i, h_{i-1}, \ldots, h_{i-n}]$
Linear ($\Delta t = 1$)	$\left[2, -1\right]$
Linear	$\left[\left(1 + \frac{(t-t_i)}{(t_i - t_{i-1})}\right), -\frac{(t-t_i)}{(t_i - t_{i-1})}\right]$
Quadratic	$\frac{1}{t-t_i}\left[\left(1 + (t-t_i) + \frac{1}{2}(t-t_i)^2\right), \left(-(t-t_i) - (t-t_i)^2\right), \frac{1}{2}(t-t_i)^2\right]$
Catmull-Rom	$\left[\frac{5}{2}, -2, \frac{1}{2}\right]$
Richardson extrapolation ($\Delta t = 1$)	$\left[\frac{7}{3}, -\frac{5}{3}, \frac{1}{3}\right]$
Richardson extrapolation	$\left[\left(\frac{4}{3(t_0 - t_1)} - \frac{1}{3(t_0 - t_2)}\right)(t-t_1), \left(-\frac{4(t-t_1)}{3(t_0 - t_1)} + 1\right), \frac{(t-t_1)}{3(t_0 - t_2)}\right]$

Table 1: *FIR-filter coefficients*

In equation 2.37 above, we have just shifted the old input signals one step further in the vector and added the new input signal to it. Then we have to apply our coefficients that make our filter unique.

$$y = \begin{bmatrix} h_0 & h_1 & \cdots & h_n \end{bmatrix} \cdot \bar{x}_t \qquad \textit{(eq. 2.38)}$$

The problem is to find the coefficients h_i. If we are able to do so, then we can extrapolate with different algorithms such as those described above. In table 1 above we have summarized the results of the coefficients and algorithms that we have described in this section.

2.3.3 Tradeoffs

When it comes to actually choose a method that is suitable to use in a computer game, we have to consider the variables in the environment and how the objects actually move. A rule of thumb is that, the faster the response in a network is, i.e. the shorter the roundtrip latency is, the less the importance of choosing a prediction algorithm is. In a network which is stable, i.e. one that does not have much packet loss, we could make more use of algorithms that are more dependent of a uniform time step, such as the Richardson extrapolation method. In chapter 4.3 we will get results from simulations and discuss this topic in more detail.

When the delay increases we have to be careful when we are using polynomial equations. Consider a simple function like $f(x) = x^2$. If we have got this equation from the last three measurements, $t_{i-2} = 1$, $t_{i-1} = 0$ and $t_i = 1$, then it is very likely that we will get a good extrapolation value in the next time step, t_{i+1}, which should in this case result in $t_{i+1} = 4$. But because we use a quadratic expression, we can easy imagine how fast this could turn from being a good, to a very bad guess. As a side effect of that the distance has a quadratic growth is that the object also will get a higher velocity. And as a result we will have objects that start slow at every new state update, and then accelerate until we get a new state update, only to slow down again.

The choice of prediction algorithm could and should also be based on what kind of objects in a game it is supposed to be used with. An object that follows the laws of physics can and should be modelled and predicted by the use of these laws. To predict the trajectory of a football is a good example of this.

3 Architecture

In this chapter we will discuss architecture and design of computer games. We will first start with an introduction to the asynchronous network framework called Twisted. This is followed by a brief description of different models. We introduce the concept of predictors and replicators that we are using in the shuffle puck game. We end this chapter with discussions about how bandwidth, time synchronization and framerate of games will affect the playability of the game. This chapter is required to be able to understand how shuffle puck is implemented.

3.1 Twisted

In this section we briefly discuss how Twisted works. This section goes into some of the features of Twisted that are used in the test application, shuffle puck, which is discussed both in this and the next chapter.

3.1.1 Overview

When it comes to writing networked applications, it often seems that programmers more or less write the same code for networking, but with small adjustments depending on the application. Twisted offer APIs to the programmer for different kinds of protocols. There are protocols such as HTTP, FTP, DNS, IMAP4, and SMTP included in Twisted, so if the programmer wants his application to use an FTP-connection, he would just use the FTP protocol from Twisted.

Twisted's abstraction level is high. When writing an application that use TCP, there are methods that is called when something happens in the network. Basically programmers write a protocol which should know how to make an incoming connection and how to behave when loosing a connection. It should also be able to know how to send and receive data within an application. The high abstraction level makes it both easier and faster to develop programs.

3.1.2 Asynchronous networking

When writing networked applications the developer has some different approaches to choose between. He could choose to spawn a new process for every connection, he could choose to handle each connection in a separate thread or he could choose to manage all the connections in one thread. Spawning processes in operating systems takes a lot of the computer's resource, so that's generally not a good idea when it comes to handling network connections. Threading is not very expensive, but with threading comes other problems, like handling shared data, i.e. preventing deadlocks etc. The last of the above proposals, manage all connections in one thread, often leads to an implementation where functions is registered and is called upon events. That is more commonly known as asynchronous or event-driven programming. That is the way Twisted handles connections. Twisted have an event loop called reactor. It is the main loop that runs all the time the program is running. It's the reactor that listens to the

network and delivers the data to the relevant protocol object. The reactor maintains this in a non-blocking way.

3.1.3 Protocols

A *Protocol* in Twisted is the class that handles incoming and outgoing data. It is the reactors responsibility to serve the protocol with this data. When a developer implements a new protocol he just subclass the *twisted.internet.protocol.Protocol* class and overrides the methods in it with his own implementation. An implementation of a game server protocol that responds with a welcome message when a player connects, and handles incoming data for a game server could be implemented in Twisted like this:

```
from twisted.internet.protocol import Protocol
  class gameServer(Protocol):
    def __init__(self, serverName):
      self.name = serverName

    def connectionMade(self):
      Print "Client connected"
      self.transport.write("Welcome to our game server")

    def dataReceived(self, data):
      # Process the data
      self.processedData = processData(data)
      # Respond to client with name of the server and processed data
      self.transport.write(self.name + self.processedData)
```

Figure 11: *A server protocol in Twisted*

The server responds to the client with the *self.transport.write()* method. It's the transport that represents and wraps the physical connection of the framework which is talking to the network.

3.1.4 Perspective Broker

The concept of Perspective Broker, abbreviated PB, is that a programmer may write programs that can serialize objects over a network connection and there is also the possibility to do method calls on remote objects. PB reminds of the RPC technique, but there are some differences when it comes to how a programmer can access remote objects. Twisted uses a method that depending on what prefix the programmer gives a remotely accessible method provides different kinds of access and views of the objects. The easiest way is to subclass *pb.Root* for doing the object remotely accessible and gives the name of the methods the prefix *remote_*. If it is important to be able to tell who is calling the remote method the programmer could do this by adding another prefix, *view_* to the methods. A programmer could mix the prefixed methods with arbitrary method names to make a distinction between what is allowed to call remotely and what is not. This is the main difference between RPC and Twisted's PB.

When it comes to return objects i.e. copy objects so a local copy will be available, Twisted has two facilities. It can copy the entire object structure or it can keep a cached version of the object and only update it with new information. When copying objects, i.e. returning them from methods or using them in arguments, the objects are serialized before they are sent. And when the caller receives the object it is unserialized again.

Due to security reasons it is not possible to just copy every kind of object. A programmer has to define which objects are allowed to be copied. Standard primitives such as integers, dictionaries, strings and so on are an exception of this. The security restriction exists because it must not be possible for an outside program to get access to important data and programs. If it were possible, a remote object could get access to the whole file system for example and read secret information.

3.1.5 Deferred – A delayed object

As mentioned earlier, Twisted is an asynchronous non-blocking framework. Twisted solves the non-blocking part with something called Deferred. When an application is waiting for data it is not allowed to stop executing the application in a non-blocking environment. If for example an application is doing a remote method call and the network traffic is going slow it could use the CPU to perform other actions instead of just be idle in the meantime. In situations like this Twisted returns a Deferred instead. Programmers can attach a callback function to a Deferred, and as soon as the result is available to the Deferred, the attached callback function will be executed. A Deferred always returns a result. If something goes wrong an error is generated. This error can also be registered together with a callback function that executes when an error has occurred.

Multiple callbacks can be added to Deferred. The result of the preceding callback is given as argument to the next callback. This makes it possible to chain method calls in a matter similar to pipes in UNIX [11].

3.2 The model and its purposes

When it comes to making a multiplayer game you have to decide which model you are going to use. Is it a client/server solution where the clients run all simulations in their own applications and the server only acts like a message replicator? Or should the client be as thin as possible and let the server handle all calculations? In both of these models the communication is only between the server and the client, i.e. there is no communication at all directly between the clients. We could also think of a model that is implemented as a peer to peer network, where all clients talk to the other clients and run a complete simulation of the whole game. In the shuffle puck application that has been developed we have chosen to go for a client/server solution where the server is the one that has the authority regarding all the participants in the game.

The main purpose of shuffle puck is to be able to test how both known and unknown algorithms work in practice. The algorithms themselves most often have various behaviours depending on parameters in the game. Parameters that affect their behaviour could for example be clients' framerate, the servers' framerate, how often packets are sent, what kind of movement pattern the objects have etc. In this application it is possible to tune parameters like these and study the behaviour according to that.

3.3 Predictors and replicators

In shuffle puck we have implemented something that we call predictors and replicators. The clients in the game have one unique predictor for every object in the game and it has got a replicator for the object that it controls, i.e. the paddle. The server has both predictors and replicators for all the players in the game. For the ball it only has a replicator. The basic idea with a replicator is that it gets a value, and then it replicates that value as soon as possible to all the other participants that listen for those kinds of values. In the shuffle puck application a client for example sets a new position- or movement-value into the replicator that belongs to the paddle. The replicator immediately put this value into Twisted's send method, and the value is sent to the server. When the value reaches the server it is put into the predictor that represents the client who was the source of the value. Sending values from server to client is the same procedure. The ball as we stated earlier only got an associated replicator at the server. This is because we used the client/server model where the server is the one that has valid information about the game and the ball isn't owned by any player. So the values of the ball object are directly read by the server and then put into its associated replicator and the rest of the procedure is the same as in the example above.

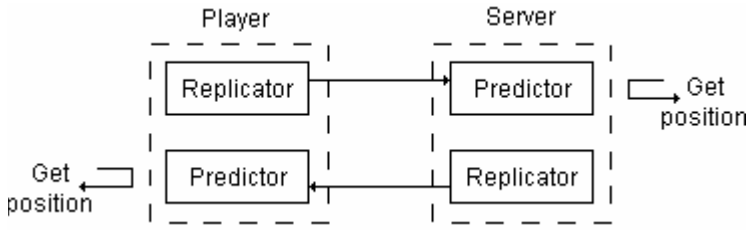

Figure 12: *Illustration of predictors and replicators*

By using predictors we gain advantage when it comes to evaluating how different prediction algorithms work in practice. We can just exchange one predictor with another predictor that uses some other prediction algorithm. But we have to bear in mind that in some cases, like the clients' paddles, we use two predictors from that we have sent the movement to that we render the result of the movement. This introduces more uncertainty than it would have if we used only client side prediction of the objects in the game. This uncertainty can be reduced by the server only using predictors that give the last value that was set to the predictor. Due to this reasons it is not possible to change predictors at the server in the shuffle puck application.

3.4 Bandwidth usage and communication

Because this thesis deals with delays and how to synchronize virtual worlds and not how to minimize bandwidth usage, we have not put any effort in reducing the bandwidth in shuffle puck. Usually developers design their own application protocol and pack messages into bytes which are sent between the involved participants. In shuffle puck we use message objects for communication. There are different kinds of message objects for the actions in the game, depending on its purpose. All messages that are used are inherited from a base class Message. That base class contains a timestamp of the actual time when the object is instantiated. Because the messages that

are to be sent are objects, we have to serialize them, and this is done by using Python's pickling concept. Pickling in Python yields either an ASCII string or a binary representation of the object.

3.5 Time synchronization

When both the client and the server run on the same computer, the clock they are using is the same. This makes the clocks perfectly synchronized. When running at different computers the result will be incorrect. To get this to work, a time synchronization protocol has to be implemented. It is possible to use shuffle puck on different computers, but there is not any time synchronization protocol implemented, so the result will be useless if doing so. There exist a few time synchronization protocols today that could be implemented to solve the problem.

3.6 The impact of different framerates

The framerate in a game is usually not a fixed value: it constantly gets higher, and then suddenly drops again. The main reason for framerate drops is when a game is rendering a lot of objects or is doing some other heavy calculations that takes a lot of CPU power. Dropping framerates affects the prediction, mostly due to that the algorithms often use the time, Δt, that is the time that has elapsed between an event really occurred and to that the event is evaluated. A greater Δt usually results in an increased prediction error. Shuffle puck has a quite stable framerate, mostly due to so few objects involved in the game. To be able to evaluate behaviour at different framerates, we have implemented the possibility to change the framerate on demand, both for the clients and for the server.

When we are lowering the framerate for the server, problems occur with collision detection. For example when the ball moves more pixels than the width of the players' paddle in one frame, then it is possible that no collision occur at all. In shuffle puck the width of a paddle is 20 pixels, so when the framerate of the server is lower than 20 frames per second we are starting to get this kind of problem. The problems them selves can be solved by using better collision detection algorithms, but no such algorithms are implemented in shuffle puck.

4 Evaluation

In this chapter we will describe the details of how the shuffle puck application, that has been mentioned several times in this report is, has been implemented. We will get numerical results of the linear, quadratic, Richardson and Catmull-Rom extrapolations methods in conjunction with linear and Catmull-Rom interpolation. These results have been generated with the Cursorsimulation utility, which will be described in this chapter. We have a section about how commercial games deal with the latency and jitter problems. And we end this chapter a summarized conclusion of the whole project and some proposals of what could be done in a further thesis on this topic.

4.1 Introduction

To be able to evaluate this project we have used three steps. First we have studied the theory, mostly interpolation and extrapolation methods. Then we have tried to combine the theories to get a good solution of the prediction problem. And to be able to see the how the algorithms works in real time we have implemented the shuffle puck game where we could check if the algorithms really worm as good in practice as some of them are told to do in theory. To get valid results we have also implemented a utility called Cursorsimulation to do simulations for us.

4.2 Shuffle puck – a prototype

As stated in the previous chapter, shuffle puck is a client/server solution. Each client is supposed to be a light weight version of the server. But due to that the server is the authoritative of the game and does not need to render objects we have to make different versions for the involved objects, depending on if it is an object in the server or in the client. In the class diagram, figure 13, we can see that we have solved this by subclassing the *Ball*, *Bat*, and *Network-Server/Client* classes. The similarities between the server and the clients make the class diagram appear mirrored.

4.2.1 Network handling

When we start the server it will first initialize all the objects with default values. The server has network objects that it will use for communication with the other clients. These objects are of the class *NetworkServer*, which is listening for and handling incoming UDP messages. Those messages are unpickled so they may be handled as real objects of the subclass *Message*. If a message is not recognized an exception will be generated. This could have happened e.g. when an UDP packet got lost.

Messages and network handling that is not time critical is handled by a TCP connection. For this we have implemented communicator classes. In the class *Communicator* we have subclassed Twisted's *pb.Root* class, i.e. we have used the remote object concept Perspective Broker that we discussed in chapter 3.1.4. In shuffle puck we use this course of action to change the frame rate of the server and to connect the clients to the server.

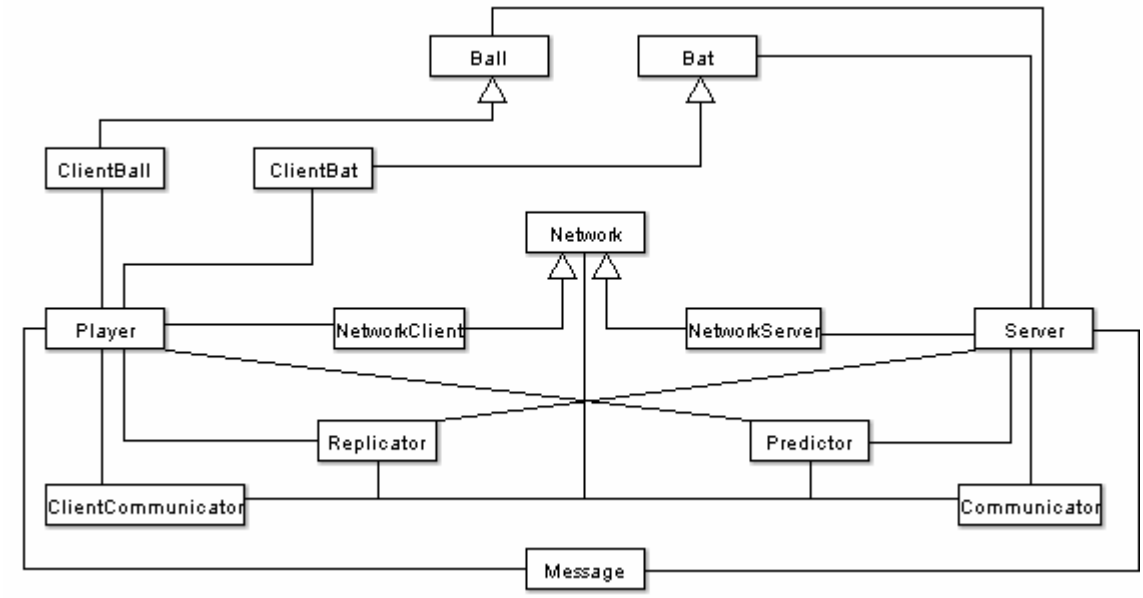

Figure 13: *Class diagram of shuffle puck*

When a client starts its application it would not get connected to the server automatically. The client will initialize the network objects in the same way as the server does, but the client does not have any methods that the server can call. When the client tries to connect to the server it first tries to get the access to the server's *Communicator* object and call the server's remote connect method. Again we are using Twisted's deferred concept for this. If everything is working as it is supposed to do the server accepts the client and adds it to the game. If something went wrong the deferred will call an *Errback* method instead.

When a connection is initiated between a client and the server they start to send messages to each other. UDP messages are not sent directly by the client object or the server object. Instead UDP messages are sent by the replicators that are connected to the objects in the game (see chapter 3.3). The client only knows about the server, so it only has to send messages to the server. In addition the server knows about several clients and therefore has to send messages to all of them. When the server receives a message it updates the predictor that is associated with the sender's replicator. The client uses the same approach when it is receiving messages. A client checks the origin of the message: if the origin is unknown, then it is a new player that wants to participate in the game, and then the client adds this new client to its data structure of the players that are participating in the game.

4.2.2 Data structures

Python is a very nice programming language when it comes to using lists and dictionaries. It offers a lot of possibilities to access unique elements and subsets of elements. You can also, for example, put an object of an arbitrary class, an integer and a string in the same list. This freedom can make development both faster and easier than

it would have been in some other programming language where you have to use the same type in lists, like Java. But it also puts demand on the developer to be structured and to know what he is doing. In shuffle puck we have used dictionaries to store the *Player* objects in both server and client. The key in those dictionaries are a *tuple* of the player's IP address and its port number. By using this approach we can have more than one player from the same IP.

The predictors uses ring buffers to store the last *n* messages. The messages are stored as *PlayerState* objects. The underlying data structure for the ring buffer is an ordinary Python list. When a new message arrives to the predictor it will update its ring buffer with the new message and delete the oldest in it. The *PlayerState* objects are used to be able to get information about the variables in the previous update states. We could get information about what time the message was constructed, what the position, velocity, acceleration and angle were at the actual time etc.

4.2.3 Graphic handling

All the graphics are rendered using *Pygame*, which is a set of modules designed for writing multimedia applications and games in Python. Pygame also supports input handling from users. In shuffle puck we have two clients running in the same Pygame window. This has been the source of some problems during development. The main loop of shuffle puck runs as fast as the computer allows it to do. Then the two clients run and render their applications respectively. The clients can have different framerates. This is accomplished by just letting a certain amount of time elapse, until each client does a loop in the game. Objects in the game are updated and rendered with new positions that are the positions that the predictors give to them. So what we actually see at the computer screen is the values that predictors give to us and not any internal computation of the objects positions.

Collision detection is achieved by using Pygames' *Rect.colliderect(rectstyle)*. What this method does is that it checks if two rectangles overlap each other. To be able to decide from which direction we got collision between the ball and a paddle we have made four very small *Rects* inside the rectangle that represents the player's visible paddle. The small rectangles are located at the edge of all four directions of the paddle. Collision detection in shuffle puck is achieved in two steps. First we check if two rectangles overlap, i.e. if the ball and a paddle overlap. If that is the case, then we go further and test which of the other small rectangles the ball does collide with. When we know this it is easy to change direction of the ball. This way to do collision detection works fine as long as the framerate is so high that the ball can not travel through the paddle. If that is not the case, then we get the problem that we described in chapter 3.6.

4.2.4 Message objects

A *Message* object is a superclass for the messages that are sent between the clients and the server. As soon as we instantiate a *Message* object the time is saved. There are different subclassed messages depending on whether it is a message from the server to a client or it is a message from a client to the server. For example it is only the server that

sends out the ball position and therefore it's only needed by the server. Earlier we said that messages are pickled, i.e. made to an ASCII- or binary representation of the objects. It is the subclassed *Message* objects that we are pickling before they are sent over the network, and they are unpickled when they are received on the destination side of the network.

4.3 Evaluation of extrapolation with interpolation

In this section we are going to present results of how four extrapolation methods are working together with two interpolation methods. To get the results we have written another Python program called Cursorsimulation, which gives the user the freedom to draw any type of time stamped curve. In Cursorsimulation we can simulate any type of curve, i.e. any type of steering behaviour that we could imagine a player to do.

4.3.1 Extrapolation methods

The extrapolation methods we have used are linear-, quadratic-, Catmull-Rom- and Richardson-extrapolation, i.e. the extrapolation methods that have been discussed in chapter 2. All four methods use the FIR-filter concept, with the coefficients presented in chapter 2.3.2.

4.3.2 Interpolation methods

As interpolation methods we have used linear interpolation, which interpolates between the extrapolated points and a Catmull-Rom interpolation, which also makes use of the points in the history. Interpolation with Catmull-Rom gives smoother paths than using linear interpolation, but as a side effect it can gain bigger deviation from the true path, resulting in a greater average error.

4.3.3 Cursorsimulation

In Cursorsimulation we draw the curves that we are going to use in the simulation process. When we draw a curve, Cursorsimulation stores information about the actual position of the mouse pointer and the actual time when the mouse pointer moved over a certain pixel. Because of this it is possible to simulate both slow and fast objects, and objects that move smoothly or very irregularly.

4.3.4 Time history

We thought that it would be interesting to know if we could gain advantage by taking care of the time stamps of the network occurrences in the past. By using only the last time stamp we will get a faster response when we get a deviant behaviour of the network traffic. This could lead to a greater diverge from the true path if the wider gap in the network occurrences was just the result of a missing packet. In Cursorsimulation it is possible to set how many of the time stamps in the past it will make use of, when it is computing a new predicted time stamp. The greater the usage of history, the more regular the update steps in terms of time. The time averaging technique could be seen as an averaging filter of the time steps.

4.3.5 Latency and jitter

It is interesting to have the possibility to tune the latency to see how the well the prediction algorithms will work at different levels of latencies. Most of the algorithms

perform better when we have a regular update frequency, so to get as fair results as possible we have implemented in Cursorsimulation both a latency- and a jitter-setting. The latency is static and must be the same during the whole simulation, but the jitter is multiplied by a random factor, so the outcome of using both latency and jitter is a simulated network update that will be a good simulation of a real network. Let us say that we set our latency to 50 ms and our jitter to also be 50 ms. Then we would have simulated network updates that is in the range [50, 100] ms. This is a very common scenario of network behaviour in existing games today. We've chosen to only add the jitter and not to subtract any jitter. With this approach we would not get any unnatural negative updates.

4.3.7 The test

In our test we have made three different curves, one that is subjected to acceleration, one where the path is following a zigzag motion and the last curve that is a smooth path.

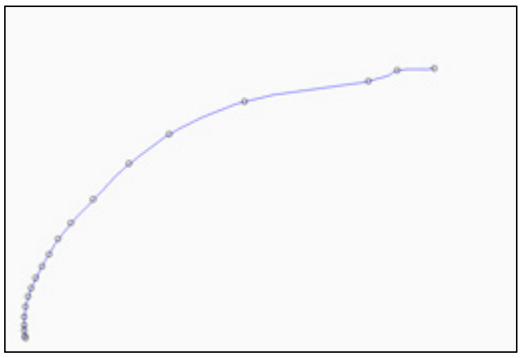

Figure 14: *Accelerated path*

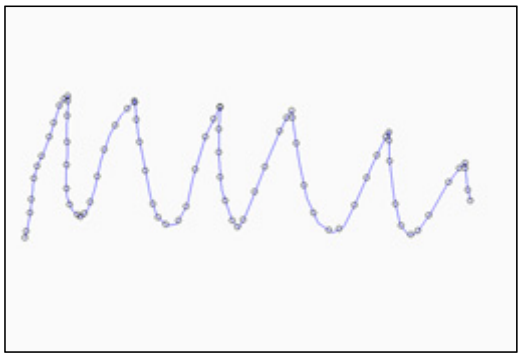

Figure 15: *Zigzag path*

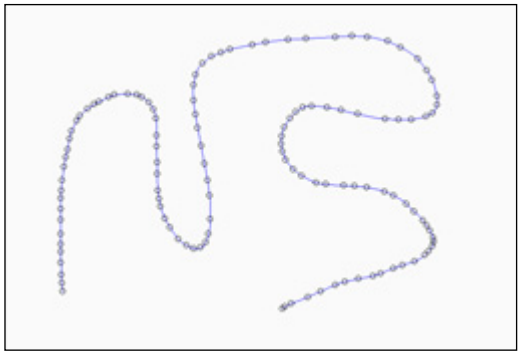

Figure 16: *Smooth path*

The black rings in the figures are an example of when network updates arrive from a server to a client. In Cursorsimulation these black rings represent these updates. I.e. this is the only moment when the client actually gets knowledge about the true position from the server. So based on these rings our prediction algorithms will first try to extrapolate where the object will be when the next update comes and at what time that will happen. When that has been done the algorithms will do interpolation between these extrapolated points. And the main goal is to have the predicted path stay as close as possible to the true path. We are going to look at these three curves and see how they work in a few different situations. We start with the accelerated path.

From the results of the accelerated path simulations we have generated scatter plots in Matlab, one where we evaluate the extrapolation methods and one where we evaluate the interpolation methods. Let us start with the plot (figure 17) of the extrapolation methods.

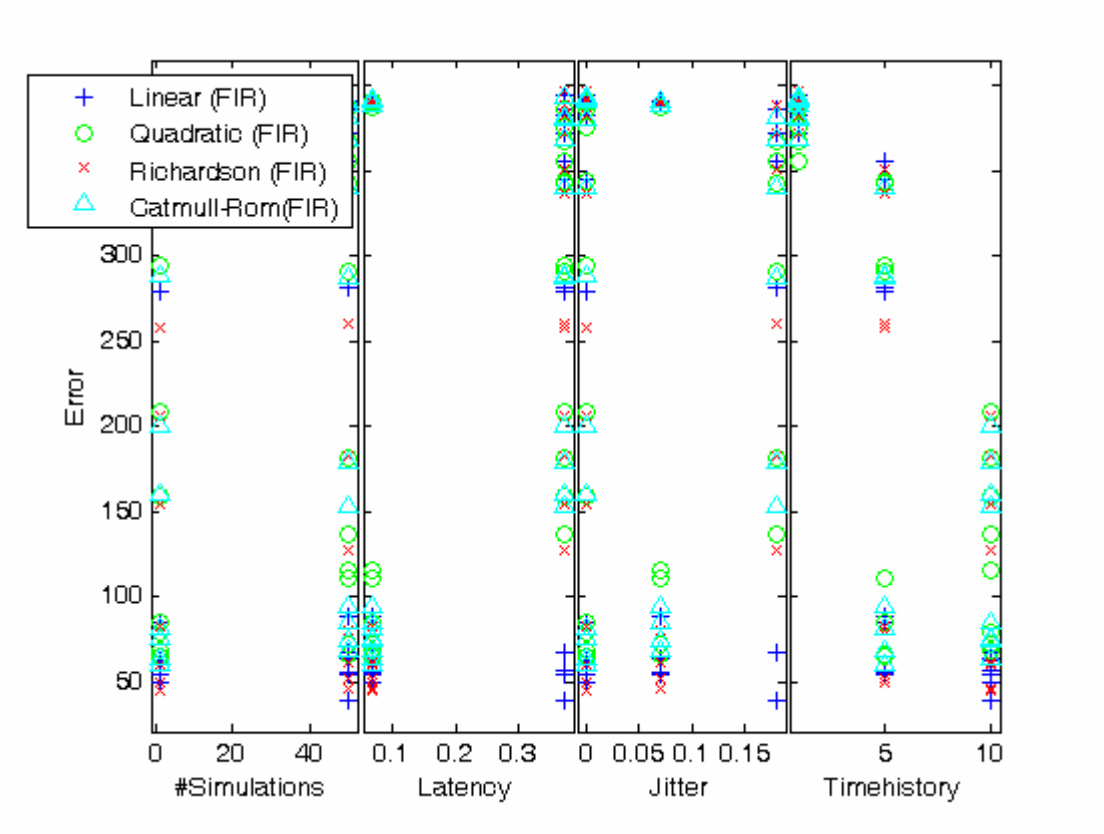

Figure 17: *Scatter plot of extrapolation methods for an accelerated path*

The first thing we notice in these plots is that there are not any particular characteristics. The measurement points are uniformly distributed independently of the extrapolation method. When we are calculating the average error (see table 2) we can see that the Quadratic, Linear and Catmull-Rom methods perform almost equally well and that the linear extrapolation method has an average error that is approximately 20 pixels lower. If we look at the latency column, we can see that lower latency gains better result than higher latency, just as expected. Stepping to the jitter column we can see we have a wide spread spectrum of results when there is not any jitter at all. The reason for this is that the Catmull-Rom interpolation method will make bad results when it's using the adjacent points that are too close to each other.

	Linear	Quadratic	Richardson	Catmull-Rom
Avg. error	212.1534	235.5711	227.4747	234.8008

Table 2: *Average error for an accelerated path*

When the jitter increases, error will also increase. The last column, where we plot the error versus the usage of the time history, we can see that the average error will be lowered and more gathered when we are only using the last known time steps.

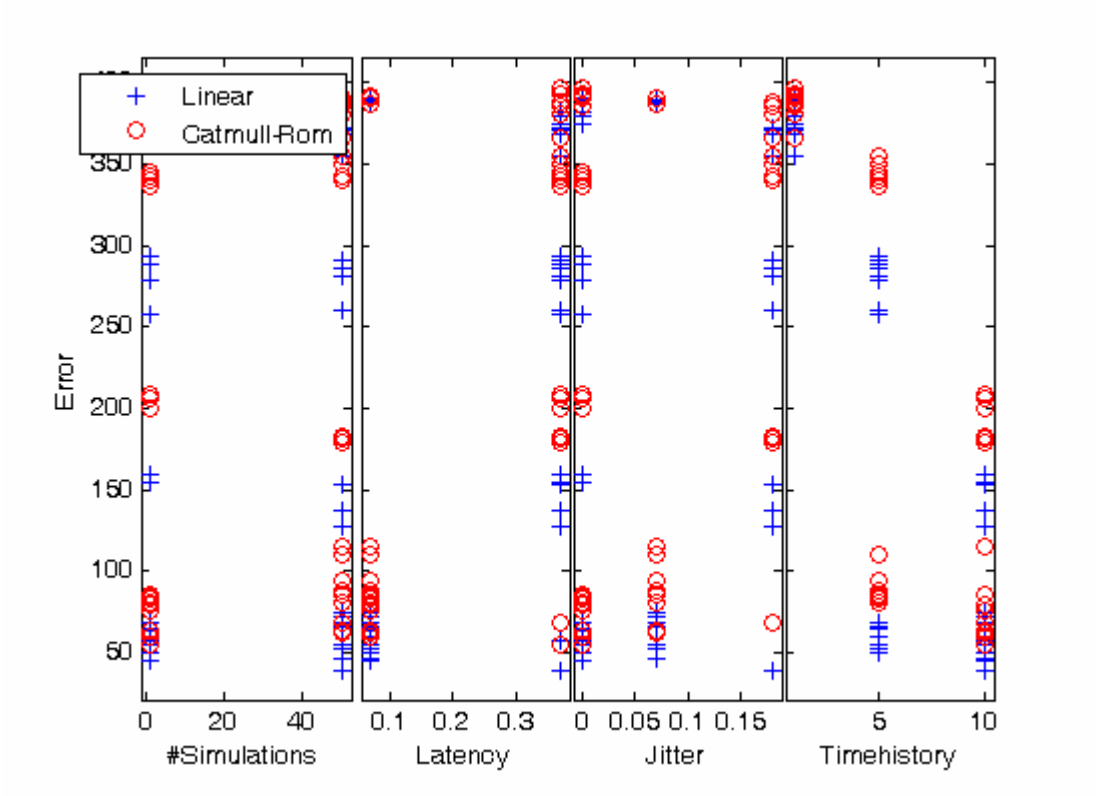

Figure 18: *Scatter plot of interpolation methods for an accelerated path*

If we are looking at the error when we are using linear versus Catmull-Rom interpolation we can again see that there is not any specific characteristics. The reasoning from the extrapolation scatter plot above (figure 18) holds for this scatter plot as well. But we can notice that the linear interpolation method has slightly better overall results in the interpolation scatter plots.

Now we will look at the plots from the zigzag path. The zigzag simulation had the exact same simulation parameters. The only thing that differed was the actual path. The extrapolation scatter plot can be seen in figure 19. When studying the picture we can see that Catmull-Rom is the worst of the extrapolation methods. This could be explained by the smooth curves that Catmull-Rom tries to do when doing both interpolation and extrapolation. We can also see that for small latencies we get almost the same result for all extrapolation methods. But when the latency is increasing the error interval error will be very widespread, from about 50 to 250 pixels. The jitter will also have widespread distribution. One thing to notice about this kind of path is that with no use of time stamp history we get a rather small error, then the error will increase and then fall back a bit again when we use more and more information of the history.

From table 3 we will see that the linear method outperforms the Catmull-Rom extrapolation method with a factor two. The other two extrapolation methods are placed in between the worst and the best method.

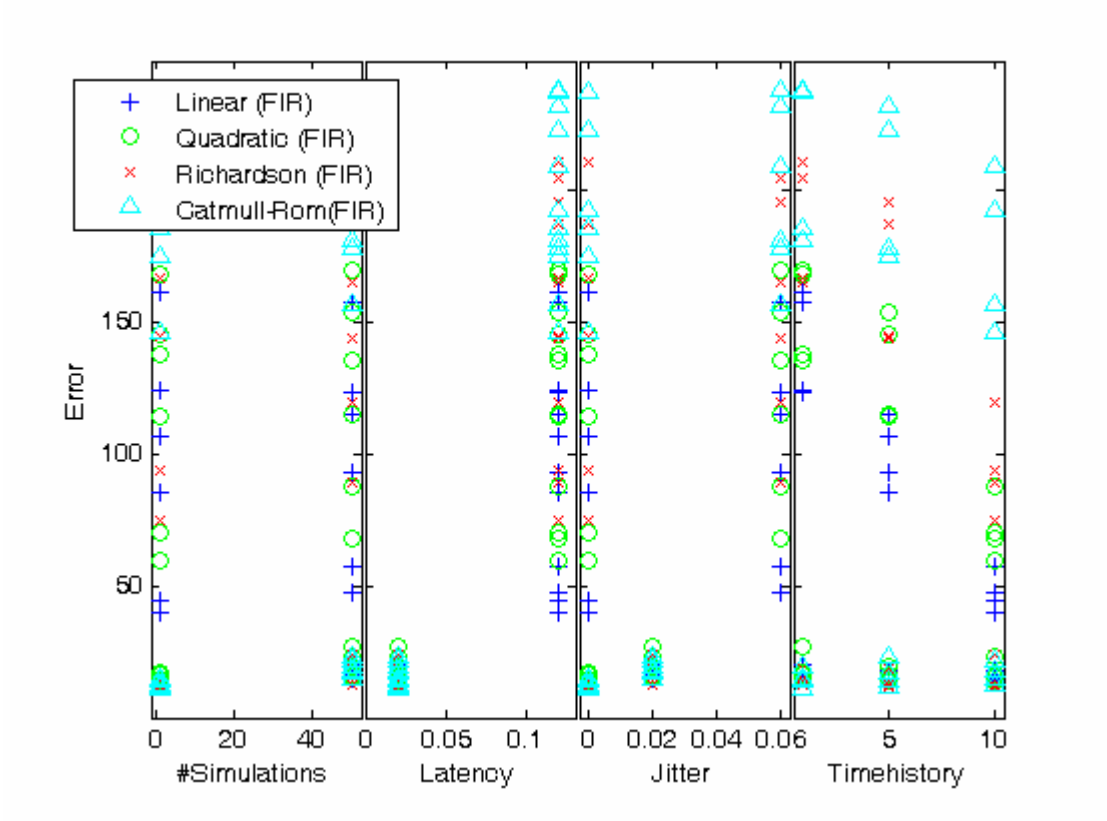

Figure 19: *Scatter plot of extrapolation methods for a zigzag path*

	Linear	Quadratic	Richardson	Catmull-Rom
Avg. error	56.5043	68.3348	82.4640	106.3748

Table 3: *Average error for a zigzag path*

In the interpolation scatter plot of the zigzag path we can verify that it is the Catmull-Rom interpolation that is involved in the bad result.

The last path we are going to look at is the smooth path. Again we have made our simulation test with the same parameters as with the two tests we have discussed before. In our extrapolation scatter plot, figure 21, we can, for the third time see e.g. that no measurement points are drifting in any particular direction, i.e. once again our distribution is approximately spread between the different extrapolation methods. In the latency column we see that Catmull-Rom extrapolation is the most gathered among the measurements when the latency is low, but when we have increased the latency the situation has changed in the other way. Now Catmull-Rom is in the top of the plot, i.e. the most error prone of them all. The jitter behaves as expected; it is more error prone when we have greater jitter in the system. We still have the same problem with Catmull-Rom interpolation and small movement steps, which can be seen in the column where we do not have any jitter.

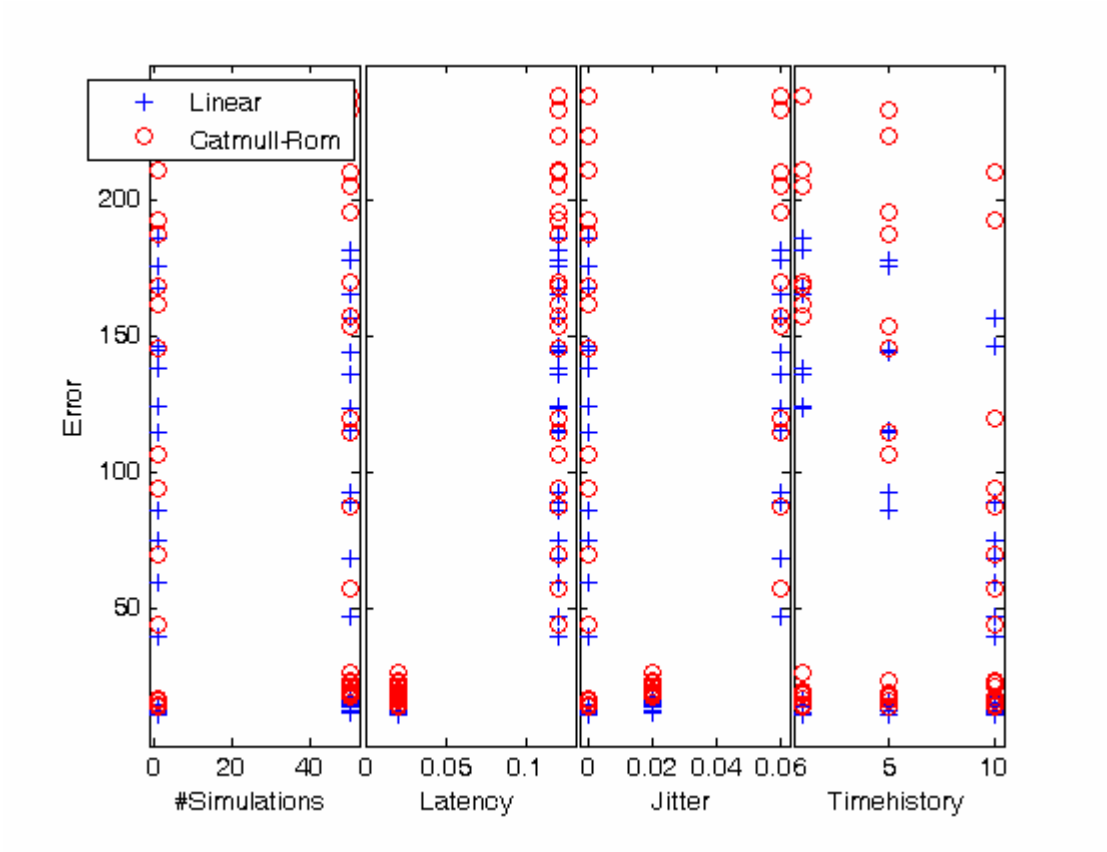

Figure 20: *Scatter plot of interpolation methods for a zigzag path*

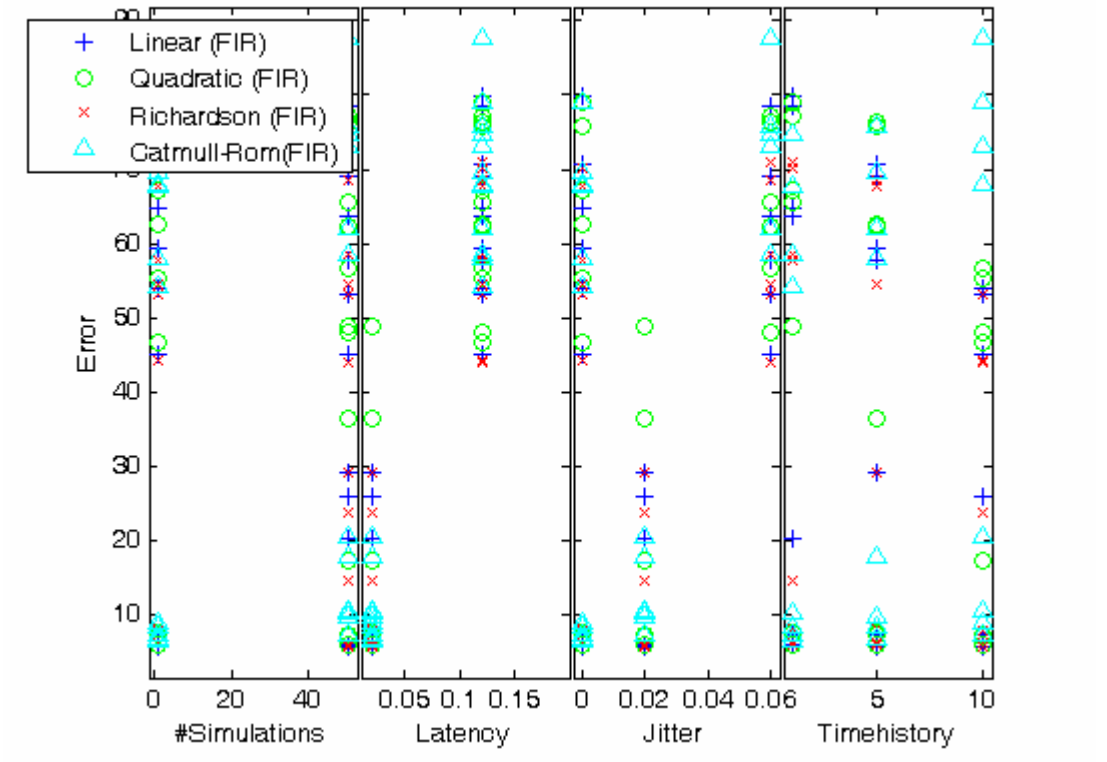

Figure 21: *Scatter plot of extrapolation methods for a smooth path*

The lower cluster is from the linear interpolation and the upper, more widespread cluster is from the Catmull-Rom interpolation. When it comes to the result of using the time history, we see that it is not as effective in this smooth case as it was in the accelerated case. This is mostly due to that the time steps are more uniform, so averaging them would not help much. From table 4 we can see that for smooth curves, Richardson extrapolation is the best extrapolator, even though there is not much difference between the different extrapolation methods.

	Linear	Quadratic	Richardson	Catmull-Rom
Avg. error	36.3881	39.0206	34.4070	39.6724

Table 4: *Average error for a smooth path*

In our last scatter plot we can again see how the two interpolation methods will perform. In this case it is clear that we will have better results when we are using linear interpolation instead of Catmull-Rom interpolation. This shows in all the columns.

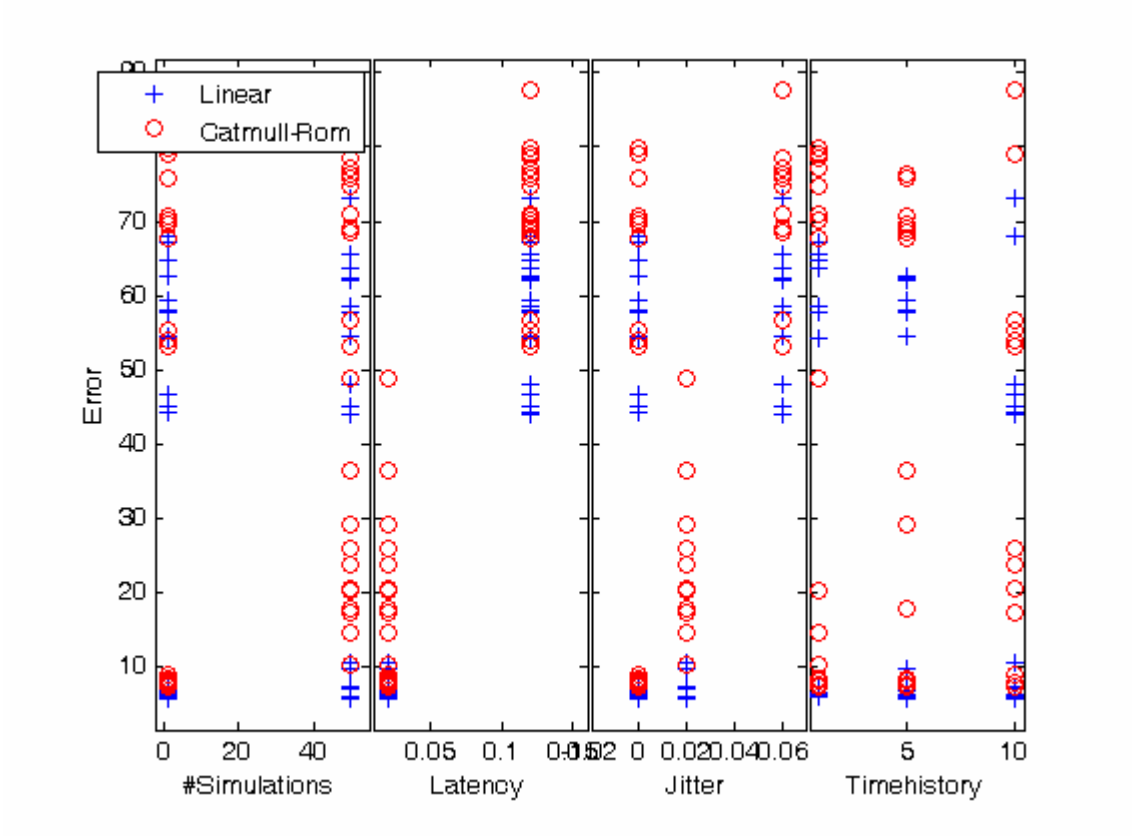

Figure 22: *Scatter plot of interpolation methods for a smooth path*

Now we have seen and discussed three different paths, all with unique features. The first was the accelerated path, where we would perform better if we were using time stamp information from the past. In the next path, the zigzag path, we saw that the linear methods, both for interpolation and extrapolation outperformed the other methods. And

in the last path, which was the easiest to follow, we did not get much difference in the different prediction methods. We noticed that we did not gain as much advantage using time history, when the time between then time steps were more uniform. The best overall prediction method was the linear prediction. But we have to bear in mind that the paths in the linear way will make straight lines between the interpolated points, and this could look strange in a computer game.

We also have to bear in mind that even though we got poor results in terms of error when we were using other prediction schemas than the linear, we could get smooth paths that followed the curve very well, but it predicted too far or too short from where it was supposed to be. So instead of getting the error by being of the path in a perpendicular way, they will get their error by predict its next position right on the path, but to far (or short).

4.4 Network handling in commercial games

Here we will discuss how two of the most well known multiplayer games do their network handling. Both titles have several releases, like for example *Q*uake that has also been released as *Quake2* and *Quake3*. This section should be seen as a general approach for all releases within a game.

Most games rely on linear extrapolation paths. This works quite good if the update interval between packets is rather small (Quake 2 uses 0.1s between the updates). Lately there has been some effort in doing prediction schemes that gain a better result than just doing linear extrapolation. Unlagged is a project that has attacked this kind of problem [12]. Hitscan weapons, i.e. weapons that hit its destination directly, have to be aimed ahead of the target when no compensation is done at all. This problem is compensated in Unlagged by remembering where every player has been, back to about a second to go. Unlagged is doing hit test against where the potential targets were seen, rather than doing hit test based on where they actually are at the moment. The technique is called "backward reconciliation": during the hit test players are moved back to where your player saw the other players when you fired. Unlagged is used in Quake3 and Wolfenstein Enemy Territory mods.

4.4.1 Unreal

Unreal uses a client server model where the server is the one which has control over the game. A subset of the true game state (that the server has) is replicated to the clients. The clients use this subset to predict the game flow by executing the same code as the server. In this way the Unreal engine minimizes the amount of data that has to be exchanged between the server and the clients.

The loop in the engine consists of as server sending updates to all the clients with information about the actual game state. The first thing the clients do in the loop is to send its requested movement to the server and as soon as the client receives a new game state from the server it renders the world and the objects in it. To take care of the problem with limited bandwidth, the Unreal engine tests which objects that is relevant

for the client to get an update about. Objects that are not visible or audible are example of things that are unimportant to send game state updates to clients. If a lot of objects are relevant, then Unreal uses a technique that prioritizes which objects that are the most important to update. Here things like movements and projectiles are considered as more important than decorative thing in the background for example. After a certain amount of time has elapsed a "tick" function is called in the engine. It is only when this function is called that the server and the clients exchange data. So a value that has changed to a new value and then changed back to the same old value within a tick would not be exchanged between server and clients.

The prediction techniques are not actually in the network code in Unreal. Instead it is implemented using scripts, known as UnrealScript. This script uses an approach that can be described as a lock-step predictor/corrector algorithm. The movement the player does is translated into physic equations that finally summarize the movement into a 3D acceleration vector. The client sends the 3D vector and the actual time in a message to the server. Right after doing so, it calls one of its own functions that move the player according to the information that has been sent to the server. The server immediately replicates this to the other clients in the game. Now it could have happened that the client could not move all the way he told the server that he wanted to do, because of the server that is the authoritative about all the objects in the game. When the server notices this, it sends another message to the client that calls a function that adjusts the position of the player. The client updates the location to exactly the position that is in the adjustment message from the server. But, because of the message is delayed the client will re-run old movements (that are stored in a linked list) that are newer than the timestamp in the update message, based of the new adjusted position [13].

4.4.2 Quake-series

In *DOOM* (which is the predecessor to the Quake-series) the networking was achieved by a peer to peer approach. With this approach every player in the game run a full simulation of the world. This approach has the shortcoming in the overall bandwidth requirement, which is $O(n^2)$. It is also sensitive for packet loss and latencies. During usage, such weakness will halt the simulation for a resync recovery procedure. In Quake, ID-software introduced a client/server solution where the client was a dumb client that only updated its own local copy of the current scene, according to the messages it received from the server. The Quake client responds to LOS (Line Of Sight) changes immediately; all other messages such as movement are sent and processed by the server. This approach introduced a delay in the players' movements. The next step was to let the player to be able to respond to movements directly. This concept is called POV latency compensation and was introduced in QuakeWorld. When this was introduced there was a need to make the client smarter. For instance, there had to be some client side collision detection, so that a player was not able to run through walls etc. Quake 2 used the same approach as QuakeWorld, but had more client side handling, which mostly consisted of subjects related to graphic rendering.

All the networking is achieved by using UDP, both reliable and unreliable packets. The reliable messages are queued until the client gets a confirmation from the server that it has received the last reliable packet. The server uses sequence numbers to be able to decide if it got all the reliable packets or not. If a reliable packet gets lost, then the next

packet will include not only the messages that were lost, but also more messages that eventually have been added to the reliable queue [14].

In Quake 3 ID-software used another approach when it comes to network handling. They replaced the previous network packet structure with a single packet type. This new packet holds the information about the clients' necessary game state. The game state that the server sends is a delta compressed packet that is made from the differences from the last state that the client has acknowledged and the true state that the server is processing. Extrapolations are then done, based on the information from the information in the last state. The good thing with this approach is that the server never has to wait for an acknowledgement and the overall latencies are much lower. The flaw is that it is more bandwidth consuming because of that the server is constantly sending out new packets instead of just sending new packets when it did not get acknowledge messages from the clients. Another drawback with this approach is that the server has to buffer a lot of data. It has to keep track of which states each client has acknowledged and buffer all the other states that the clients has not responded to [15].

4.4.3 Pros and cons using the techniques in existing games

Linear extrapolation has both benefits and drawbacks. The benefits are that even though the algorithms do not produce the correct result, it would not get too bad either. Consider using a quadratic polynomial, it can be good in a small interval, but it can very quickly get really bad, just because of that it's quadratic. So when update intervals are small, then a linear path will produce quite good result. The drawback is that it doesn't make use of old information. This will give bad extrapolation in curvatures. Here a higher order polynomial would get a better extrapolation polynomial that would follow the curve better. The Unlagged method makes use of a history queue of the players' states, up to 17 states which are about 850 ms of a history. By going back in time to see if a player aimed correct at a target makes a fair decision if he hit his target or not. But the result is that the targeted player will get a feeling that he was shot behind walls, i.e. the same problem as we got without latency compensation, but with the difference, this is fair for all participants, i.e. it do not just favour the one with low network latency.

4.4 Future work

To be able to use a Kalman filter in the way it is supposed to, our noise has to be Gaussian noise. The lack of this information in a computer game makes our approach to do Kalman filtering slightly incorrect. So an investigation of a better way to estimate the Gaussian noise could make the whole prediction better. This could maybe be precalculated depending on the type of the game, or by using some steps in the past time to calculate the error depending on the deviation of the object.

Because we only implemented shuffle puck to use a single prediction method a time, we can get the problems like the one explained in chapter 3.3. We would probably get a better prediction result if we used some kind of adaptive approach to choose prediction method by the application itself. It could for example be that we started our prediction by using some more advanced approach, like prediction with a polynomial for a certain

amount of time. When a time threshold has elapsed the application automatically changes to another prediction algorithm.

The concept in shuffle puck to use predictors and replicators could have been enhanced to implement intelligence in the program, so it could automatically adjust the packet flow between the participants. For example, when the application thinks that it is easy to predict an object, then the predictors tell the replicators that they does not need a high rate of packets, and vice versa when the application has problems doing prediction for an object.

When we are doing smoothing between extrapolated points and especially when we are enforcing C^1 continuity we can under some circumstances get wide curves. This will give us a farther distance for out object to travel and that leads us to a higher velocity. This is the problem related to the arc length of a curve.

A time synchronization protocol could be implemented so that the application could be tested in a real networked environment under real circumstances.

4.5 Conclusions

Prediction is a really hard problem to solve. We do not know when we get a new message from the network that updates our state variables. And we cannot be really sure where the object's position will actually be when we get the next update. These two sources of uncertainty is the major problem. When we have succeeded fairly well with our extrapolation step we got a new problem. Should we just change to that position, or should we try to smooth out the movements? This is often a question of how far we have deviated from an accepted threshold value. If the distance is too great, then it may be better to just set the new position instead of trying to interpolate to it. This would of course result in an object that snaps to a position, but it could be worth doing like that in a worst case scenario.

Even though consumers get Internet connections with more bandwidth available, the problem with prediction is something that still is interesting. As we have seen in this report, the bandwidth is only a small factor of the problem. The main problem is the delay in the network. And we will always have latency in networks. The speed of light limits how fast a packet can travel between computers. It takes about 200 ms for the light to travel around the world. So doing prediction is necessary.

Dead reckoning is the most widespread approach to solve the prediction problem. This is probably mostly due to that it is an easy, straight-forward idea, which is easy to implement and with relatively good results.

We have also seen that ideas and approaches from control theory also be useful in computer games. Further investigation of regulator structures like P, PI and PID would probably give us more knowledge of how to use these concepts in a computer game.

Throughout this report we have discussed and returned to a few extrapolation and interpolation methods. These methods have been discussed both in a theoretical aspect and from a practical view. In the theoretical part we have derived FIR filter coefficients, so that we could use the same method for doing extrapolation by just changing the coefficients. There are many mathematical ways to get the derivatives at specific coordinates in curves. By discretizising such functions we get the FIR coefficients, and they will works almost equally good independently of what steps we have to do to get the derivative. In Cursorsimulation we have evaluated and verified some parts from the theory. What was somewhat surprising was that the linear methods performed the best throughout almost the entire test, both when we were doing extrapolation and interpolation. But just because of that we should not drop the other algorithms. As stated before in this report we should try to make use of the laws of physics as much as possible when we have a chance to do so. If we develop a baseball simulator, then we could predict the trajectory of the ball by using the law of the physics. The result would be very smooth and with almost no error at all.

Our solutions and ideas fit as a complement when there are very temporary delay spikes in the network, and using other strategies when the network is working as it should do. For instance, one could use a simple extrapolation algorithm and do dead reckoning comparisons between the server and the clients. If a client (or the server) suddenly notices that it has not got any updates from the other, then a more advanced prediction schema, preferably with use of an algorithm that use C^1 continuity.

By developing the shuffle puck game we have tried the concept of predictors and replicators in an asynchronous environment. It works, but the overall result is not near the result of using dead reckoning with client side prediction, which is the most common way of doing prediction in computer games. The main problem is that a player can not affect the motion of a player as long as the network traffic is not flowing. If the delay gets too big, then the player will get the movement from the predictor, which in turn does the prediction based on what algorithm the actual object use. This can also make a player's object to move even though he is not pushing his object in any direction. This can happen if a player starts moving an object, then at some time before he has released his move button the network latency grows big. If he releases the move button, the corresponding message would not reach the moving object's responsible predictor. The end result is a moving object even though it is not intended to move.

As a final word, prediction schemes can be useful in networked games, but they cannot reduce the latency directly. But we can use prediction to reduce the impact of the latency and the jitter.

Appendix A

A. 1 Coefficients of Newtons' interpolation polynomial

We stated that the polynomial should be of the form

$$P(x) = C_0 + C_1(x - x_1) + C_2(x - x_1)(x - x_2)$$

We can determine coefficients C_0, C_1 and C_2 of the requirement

$$P(x_i) = f_i, \quad i = 1, 2, 3$$

We get this equation system

$$\begin{aligned} C_0 &= f_1 \\ C_0 + C_1(x_2 - x_1) &= f_2 \\ C_0 + C_1(x_3 - x_1) + C_2(x_3 - x_1)(x_3 - x_2) &= f_3 \end{aligned}$$

Solving for each coefficient we get

$$\begin{aligned} C_0 &= f_1 \\ C_1 &= \frac{f_2 - f_1}{x_2 - x_1} \\ C_2 &= \frac{f_3 - f_1 - \left(\frac{x_3 - x_1}{x_2 - x_1}\right)(f_2 - f_1)}{(x_3 - x_1)(x_3 - x_2)} \end{aligned}$$

A. 2 Richardson extrapolation

Central difference is written as and can be evaluated with Taylor series as below

$$F(h) = \frac{f(x+h) - f(x-h)}{2h} \underset{\text{Taylor series}}{=} f'(x) + b_1 h^2 + \mathrm{O}(h^4)$$

Calculate F with a twice as big value and we get

$$F(2h) = f'(x) + 4b_1 h^2 + \mathrm{O}(h^4)$$

By subtracting the equations one from another gives us an estimation of the truncation error of the term b_1h^2.

$$F(2h) - F(h) = 3b_1h^2 + \mathrm{O}(h^4)$$

solving for b_1h^2 gives

$$b_1h^2 = \frac{F(2h) - F(h)}{3} + \mathrm{O}(h^4)$$

If we put this into the expression for *F(h)* we get

$$F(h) = f'(x) + \frac{F(2h) - F(h)}{3} + \mathrm{O}(h^4)$$

and we get

$$f'(x) = F(h) + \frac{F(h) - F(2h)}{3} + \mathrm{O}(h^4)$$

A. 3 Derivation of a position

We know from Newtonian physics that

$$\frac{d^2p}{dt^2} = \ddot{p} = \frac{d\dot{p}}{dt} = \frac{dv}{dt} = \dot{v} = a$$

So integrating the acceleration twice gives us our position

$$v(t) = \int a\,dt = at + C$$

We get *C* by solving *v(t)* when *t = 0*. I.e.

$$v(0) = a \cdot 0 + C \quad \Leftrightarrow \quad v_0 = C \quad \Rightarrow \quad v(t) = at + v_0$$

Now we integrate the velocity to find the position. Again we are solving for the constant of the integration. We get

$$p(t) = \int v(t)\, dt = \int at + v_0 \; dt = \frac{1}{2} at^2 + v_0 t + p_0$$

A. 4 Hermite geometry

P_0, start point

P_3, end point

u, interpolation value over the interval *[0, 1]*

We write a cubic parametric polynomial as

$$P(u) = \sum_{k=0}^{3} c_k u^k = u^T c$$

Because of that we are only interested in P_0 and P_3 we have that $u=0$ and $u=1$, that give us following conditions

$$P(0) = P_0 = c_0$$
$$P(1) = P_3 = c_0 + c_1 + c_2 + c_3$$

The derivative of the cubic parametric polynomial gives us a quadratic polynomial that is

$$P'(u) = c_1 + 2uc_2 + 3u^2 c_3$$

So derivates at P_0 and P_3 is

$$P'(0) = P'_0 = c_1$$
$$P'(1) = P'_3 = c_1 + 2c_2 + 3c_3$$

In matrix form we can write this as

$$q = Mc \quad \Leftrightarrow \quad q = \begin{bmatrix} P_0 \\ P_3 \\ P'_0 \\ P'_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} c$$

To find the constants c_i, we solve the equations

$$c = M_H q$$

where M_H, is the inverse of M. M_H is called the *Hermite geometry*.

This interpolation polynomial using Hermite geometry is the defined by

$$P(u) = u^T M_H q$$

A. 5 Coefficients of a cubic spline

We have in a graph space that our cubic spline is written as

$$\begin{cases} x = At^3 + Bt^2 + Ct + D \\ y = Et^3 + Ft^2 + Gt + H \end{cases}, \; 0 \le t \le 1$$

By solving for each coefficient gives us

$$A = x_3 - 3x_2 + 3x_1 - x_0$$
$$B = 3x_2 - 6x_1 + 3x_0$$
$$C = 3x_1 - 3x_0$$
$$D = x_0$$
$$E = y_3 - 3y_2 + 3y_1 - y_0$$
$$F = 3y_2 - 6y_1 + 3y_0$$
$$G = 3y_1 - 3y_0$$
$$H = y_0$$

We can see that when $t=0$ we get x_0, y_0. And when $t=1$ we have to summarize all coefficients, and we end up with the last point x_3, y_3.

Reference

[1] A. Watt and F. Policarpo, 2001, *3D GAMES Real-time Rendering and Software Technology*

[2] V. Ufnarovski, September 2003, *Geometri*

[3] L. Eldén and L. Wittmeyer-Koch, 2001, *Numeriska beräkningar - analys och illustrationer med MATLAB®, fjärde upplagan*

[4] G. Welch, Department of Electrical & Computer Engineering, February 2005, *Rudolph E. Kalman* - http://www.ece.ufl.edu/publications/kalmanfilter.html

[5] D. Simon, June 2001, *Kalman Filtering* - http://academic.csuohio.edu/simond/courses/eec644/kalman.pdf

[6] G. Welch and G. Bishop, April 2004, *An Introduction to the Kalman Filter* - http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf

[7] B. Wagner and M. Barr, 2002, *Introduction to Digital Filters* - http://www.netrino.com/Publications/Glossary/Filters.html

[8] E. Angel, 2003, *INTERACTIVE COMPUTER GRAPHICS, A Top-Down Approach Using OpenGL™, third edition*

[9] N. Caldwell, February 2000, *Defeating Lag With Cubic Splines* - http://www.gamedev.net/reference/articles/article914.asp

[10] D. Lancaster, April 1993, *Hardware Hacker* - http://www.tinaja.com/glib/hack62.pdf

[11] Twisted Matrix Labs, 2004, *Twisted Documentation* - http://twistedmatrix.com/

[12] Unlagged, May 2005, *Frequently-asked Questions* - http://www.planetquake.com/alternatefire/unlagged_faq.html

[13] T. Sweeney, July 1999, *Unreal Networking Architecture* - http://unreal.epicgames.com/Network.htm

[14] B. Kreimeier, March 1998, *The Quake 2 Networking Data Flow* - http://www.gamers.org/dEngine/quake2/Q2DP/Q2DP_Network/Q2DP_Network.html

[15] Book of Hook, April 2004, *The Quake3 Networking Model* - http://www.bookofhook.com/Article/GameDevelopment/TheQuake3NetworkingModel.html