Inżynieria oprogramowania w grach komputerowych w praktyce

Hubert Rutkowski

Akademia Górniczo Hutnicza w Krakowie / Playlogic Game Factory

Streszczenie

W referacie opisuję wnioski z pracy nad wieloma różnymi projektami, omawiam dobre praktyki inżynierii oprogramowania, problemy których lepiej unikać i zalecane sposoby ich rozwiązywania.

1. Wstęp

Inżynieria oprogramowania to dziedzina inżynierii systemów zajmująca się wszelkimi aspektami produkcji oprogramowania: od analizy i określenia wymagań, przez projektowanie i wdrożenie, aż do ewolucji gotowego oprogramowania. Podczas gdy informatyka zajmuje się teoretycznymi aspektami produkcji oprogramowania, inżynieria oprogramowania koncentruje się na stronie praktycznej.

Wikipedia

Długo zastanawiałem się jak zacząć ten referat; w końcu postanowiłem podzielić się refleksją. Mam w branży wielu znajomych, kolegów i przyjaciół, często wymieniamy się doświadczeniami z pracy w kolejnej firmie. Niekiedy ich opowieści zza zamkniętych drzwi sprawiały, że włosy stawały dęba. Czasami ja dzieliłem się takimi opowieściami. Aktualnie pracuję w holenderskiej firmie Playlogic, która całkiem nieźle radzi sobie z zarządzaniem projektem, unikając błędów które widziałem wcześniej, wywołujących reakcję taką jak poniżej.

Rys. 0 Coding horror, koszmar programisty z [5]

Wniosek z tych wszystkich rozmów jest taki, że dobre praktyki inżynierii oprogramowania, są (zbyt) rzadko stosowane w firmach tworzących gry komputerowe w naszym kraju. Wynika to z różnych czynników.

Jednym z najważniejszych może być niedoświadczenie kadry managerskiej i niska świadomość tego, co tak naprawdę znaczy „stworzyć grę". Przejawia się przeświadczeniem, że gry komputerowe produkuje się *inaczej*, lub łatwiej od innych projektów oprogramowania. Wynika z iluzji, że aby doprowadzić zarządzany projekt do końca, wystarczy od dziecka regularnie grać, przeczytać kilkanaście tutoriali, śledzić nowinki w branży na Gamasutrze i Kotaku[0]... i to wystarczy.

W efekcie, ludzie pracujący w zespołach zarządzanych przez takich „fachowców", mają okazję doświadczyć pełnego przekroju „technik zarządzania": od niekompetencji, wtrącania się w decyzje podejmowane przez profesjonalistów, nagminnego przedłużania godzin pracy w nadgodzinach (oczywiście niepłatnych), do przekładania wypłat, nagłych zwolnień, niepotrzebnych konfliktów z wydawcą, anulowanych projektów i zamykania całych firm game developerskich. Stworzenie i wydanie gry jest (zbyt) często dla nas, programistów, drogą przez mękę.

W swoim referacie opiszę nastawienie potrzebne do prowadzenia projektu, rzeczy które trzeba robić / których należy unikać i podam wiele przydatnych (mam nadzieję) wskazówek. To będą moje opinie, przemyślania i doświadczenia, niekoniecznie zawsze prawdziwe - dlatego też należy je traktować z podejrzliwością a nie jak prawdy objawione. Posiadanie tej wiedzy nie przeszkadza mi w regularnym przekładaniu terminów moich

własnych projektów (na które mam zawsze za mało czasu), dowodzi to więc, że inżynieria oprogramowania nie jest banalna i trzeba do niej podchodzić z należnym respektem. Nikt nie wie wszystkiego.

Nie jesteś sam

Pierwszą rzeczą którą musimy sobie uświadomić, jest fakt, że jakiekolwiek większe projekty informatyczne powstają w wyniku zespołowego wysiłku wielu ludzi. Było to prawdą już kilkanaście/kilkadziesiąt lat temu (np. słynny IBM OS/360 opisany przez Freda Brooksa w [1]). Dwadzieścia lat temu komercyjne gry tworzone były w garażach przez niewielkie zespoły studentów-zapaleńców. W dzisiejszych realiach, rzeczywistością jest istnienie kilkudziesięciosobowych zespołów... grafików i programistów.

Czasy samotnego „programisty-kowboja (cowboy coding, [2]) tworzącego całą grę, od początku do końca samemu, z minimalnym planowaniem, bezpowrotnie minęły. Amatorskie produkcje, pisane w niewielkich zespołach, nawet te najlepsze i doceniane przez wszystkich (np. World of Goo autorstwa 2d boy), posiadają zupełnie inny „ciężar gatunkowy" i są wyjątkiem który potwierdza regułę.

Rys. 1 FarCry2 team

W takiej sytuacji jest krytycznie ważne dla powodzenia gry, aby osoby zarządzające posiadały wiedzę odnośnie sposobów działania i koordynowania dużych grup ludzi tak różnych jak programiści i graficy.

W przypadku programistów sprawa jest o tyle bardziej skomplikowana, że specyfika procesu tworzenia oprogramowania jest trudna do pojęcia dla osób o nie-technicznym wykształceniu / sposobie myślenia, zaś programiści to ludzie ~~dziwni~~ specyficzni, posługujący się niezrozumiałym żargonem. Kiedyś przeczytałem żartobliwe, lecz częściowo prawdziwe:

> I've seen occasional articles about how to manage programmers. Really there should be two articles: one about what to do if you are yourself a programmer, and one about what to do if you're not. And the second could probably be condensed into two words: give up.
> Autor nieznany

Podsumowując: gry tworzymy drużynowo. Jeden za wszystkich, wszyscy za jednego. Ważniejsze od umiejętności pojedyńczej jednostki jest zespołowe działanie... i na tym aspekcie będę się koncentrował. Niektóre z poniższych porad mogą wydawać się powszechnie znane, wręcz oczywiste... ale znalazły się tu dlatego, iż w przeszłości znałem osoby odpowiedzialne za prowadzenie projektów, dla których te dobre praktyki **nie** były oczywiste.

2. Good... things to do

2.1 Specyfikacja wymagań, określanie terminów

Obie czynności, czasami pogardliwie odrzucane jako „nudna robota papierkowa”, a częściej po prostu ignorowane lub zapominane, mają znaczący wpływ na sukces projektu. Potrzebne są, aby móc oszacować kiedy gra będzie gotowa. Dzięki temu możemy uniknąć losu Duke Nukem Forever – gry która pobiła wszelkie rekordy długości czasu powstawania. DNF jest już w produkcji 10 lat, i dalej nie wiadomo kiedy zostanie skończony, gdyż **when it's done** nie jest specjalnie precyzyjne...

Z moich obserwacji wynika, że w polskim przemyśle gier terminy są traktowane z przymrużeniem oka. Szacowane są, jeśli w ogóle, z pomocą wróżki, kart Tarota albo fusów z herbaty. Najczęściej nie określa się żadnych ścisłych terminów, czekając na rozwój sytuacji, obserwując i ekstrapolując przewidwania na przyszłość. Taki początek projektu nie jest niczym złym, jest nawet dość naturalny

Początkowo niejasne jest wiele zmiennych: jak tworzona gra będzie wyglądać, czego potrzeba aby ją stworzyć, czasami nawet kto będzie ją tworzył – proces tworzenia gier jest silnie iteracyjny i eksperymentalny. Ale jeśli po kilku miesiącach od rozpoczęcia produkcji brakuje opisanych dalej

dokumentów i liczb, może to być przepowiednia na kłopoty w przyszłości. Opisany poniżej proces jest pewną odmianą Scrum[3].

2.1.1 Początek

Podstawą jest posiadanie **design docu** (dokumentu z wizją i opisem gry), tworzonego przez game designera. Następnie na jego podstawie należy określić i wyodrębnić (*breakedown*) poszczególne *features'y* (branżowe określenie na cechy gry), ustalić kolejność ich implementacji, trudność i potencjalne ryzyko. Spisanie tego wszystkiego jest pracochłonnym zadaniem dla project managera i głównego programisty, z okazjonalnym wsparciem innych programistów, pomagających oszacować czas implementacji specyficznych zadań. Swój wkład powinni mieć też przedstawiciele innych dyscyplin.

Ponieważ zadania najczęściej będą zbyt duże aby można było realnie oszacować czas implementacji, koniecznie trzeba podzielić je na pod zadania, jak to zaraz pokażę na przykładzie. Ale często nawet wtedy określenie terminu będzie niemożliwe, gdyż wiele rzeczy będzie niewiadomą. Takie zadania należy określić „z grubsza", czy zajmie ono raczej 5 dni czy 50, a na pełne rozbicie poczekać aż to będzie możliwe.

Przykład: załóżmy że tworzymy strategiczną grę czasu rzeczywistego w 3D (RTS, RealTime Strategy). Gracz zazwyczaj dostaje możliwość przesuwania, obracania, przybliżania/oddalania widoku, zaznaczania pojedyńczych jednostek i grup jednostek, wskazywania im celu marszu lub ataku, budowania fabryk, konstruowania w nich jednostek i wielu wielu innych. Wysokopoziomowy podział zadań mógłby wyglądać tak:

ID	Nazwa	Czas tworzania (dni)	*Ryzyko*	Priorytet
1	Obsługa kamery	?	?	must have
2	Rozkazywania jednostkom	?	?	must have
3	Budowanie konstrukcji	?	?	must have
	...			

Tak naprawdę nic jeszcze nie wiemy. Dokonujemy więc podziału np. „Budowanie konstrukcji" na podzadania:

ID	Nazwa	Czas tworzenia (dni)	*Ryzyko*	Notatki
1	Interfejs do budowania w menu gry	3	małe	zakładamy że kod do obsługi UI będzie już istniał
2	Dynamiczne kolorowanie na zielono terenu gdzie można budować budynki	2	średnie	zależy od formatu poziomów i sposobu przechowywania go w pamięci
3	Odtworzanie animacji tworzenia budynku	1,5	małe	zakładamy pojedyńczą animację odtwarzaną od początku do kónca?
4	Zaimplementowanie funkcjonalności wszystkich typów budynków	6	średnie	będzie kilka rodzajów fabryk
5	Budynki działają w grze sieciowej	7	duże	
	Suma:	19,5		

Mamy teraz czarno na białym, że implementacja budynków w naszej strategii zajmie łącznie około 20 dni roboczych, czyli około miesiąca pracy. To bardzo ważna informacja. Wypisane podzadania można (i trzeba) dzielić jeszcze bardziej, ale na to będzie czas kiedy będziemy podchodzili do ich implementacji. Na razie potrzebujemy jedynie (aż) oszacować łączny czas tworzenia. Szczegółowe wyliczenia z podziałem na podzadania powinniśmy wykonać dla kodu, grafiki, muzyki i dźwięków. Ogólnie uwzględniamy czas testowania (alfa i betatesty), szlifowania, tłumaczenia na inne języki, tworzenia instalatora itd.

Sumujemy wszystko, otrzymujemy konkretną liczbę zapisaną na papierze, np. „Projekt Nightmares in the Sky, trójwymiarowa gra strategiczna = 10 miesięcy developingu". Voila! Warto dodać co najmniej 10-20% do tej liczby aby zabezpieczyć się przed nieprzewidzianymi problemami (które zawsze się pojawiają) i tym o czym zapomnieliśmy (zazwyczaj będzie tego trochę).

Nagle może okazać się, że otrzymany termin jest zbyt daleki i nie możemy sobie pozwolić na tak długie tworzenie gry. Tym lepiej dla nas. Od razu widzimy jak realistyczne są nasze oczekiwania i unikamy jakże typowego błędu, nieświadomie starając się w np. 12 miesiącach zmieścić efektywne 24 miesiące pracy, w wyniku czego to półrocze w którym okaże się, że jesteśmy daleko za oczekiwaniami, spędzamy w *crunch mode*, tzn.

poświęcając kilkanaście godzin dziennie, wliczając soboty i niedziele, na pracę aby tylko dogonić plan. Lecz wiedząc o tym wcześniej, nie stoimy pod ścianą, mamy wybór. Możemy usunąć trochę zaplanowanej funkcjonalności, uprościć niektóre z cech gry, skrócić grę, zatrudnić dodatkowy personel, przedłużyć czas produkcji, anulować projekt itp.

Dzięki takiemu postępowaniu jesteśmy w stanie również przeciwstawić się zmianom, forsowanych w późniejszym czasie przez różne osoby, z wewnątrz firmy lub VIPów od wydawcy: „*hmmm, pomyśleliście może żeby wprowadzić XXX i YYY* (dwie ulubione cechy gry tej osoby)? *Nie powinno zająć dużu czasu, a korzyści są duże i gracze to pokochają*".

Najprawdopodobniej ta osoba nie ma zielonego pojęcia na temat implementacji tego o co prosi, a że nie naciskałaby o coś mało ciekawego, najpewniej zaimplementowanie XXX zajmie dużo czasu. Po drugie, korzyści wcale nie będą takie duże, ponieważ jeśli dana cecha nie została uwzględniona lub nie została zaakceptowana podczas pisania oryginalnego design docu, musiały istnieć ku temu dobre powody. Po trzecie, są szanse, że najbardziej zadowolonym graczem będzie (tylko) ta osoba.

Widać więc, że należy ostrożnie podchodzić do „sugestii" zgłaszanych w trakcie developingu, które więcej zamieszają niż pomogą, a dodatkowo zabiorą nam cenny czas. Dysponując taką tabelką można asertywnie powiedzieć do VIPa „*okay, nie ma problemu, przemyśleliśmy tą sugestię, podzieliliśmy to na zadania i wyszło, że implementacja zajmie około 2 osobo tygodnie, w związku z czym o tyle też będziemy musieli przesunąć termin wydania gry... ale ponieważ wy nam płacicie za łączny czas developingu, możemy to zrobić. Zgoda?*".

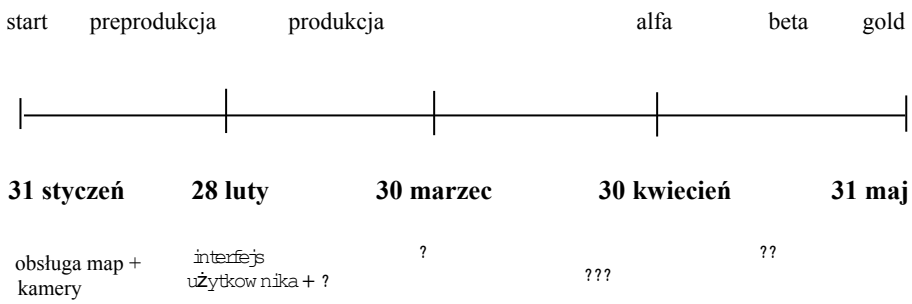
Często jednak zmiany wynikają z innych czynników i są koniecznością. Po kilku miesiącach developingu oryginalne założenia mogą się nie sprawdzić i trzeba będzie wprowadzić poważne zmiany. Dysponując wyliczeniami jesteśmy w stanie uzyskać ogólny obraz tego w jakim stanie jest gra, i lepiej zaplanować decyzje.

2.1.2 Produkcja

Następnie, aby oryginalne cyfry nie straciły zbyt szybko kontaktu z rzeczywistością, należy utrzymywać tą część dokumentacji projektu aktualną. Polecanym sposobem, jest wykorzystanie koncepcji kamieni milowych (milestones), która pięknie się integruje (a właściwie to jest częścią) *Agile sotware developement*, zestawu praktyk opisanego dalej.

Kamień milowy - w kontekście dyscypliny zarządzania projektami, jest to końcowy punkt, który podsumowuje określony zestaw zadań, bądź daną fazę projektu. Oznacza on jednocześnie pewne istotne, jednorazowe zdarzenie, które można w jednoznaczny sposób określić.
Wikipedia

Logicznym więc wydaje się podzielenie czasu pozostałego do skończenia projektu, obliczonego na podstawie specyfikacji wymagań, na kamienie milowe, z których każdy będzie polegał na zbudowaniu nowej wersji implementującej część z wymagań. Nie poleca się planować zbyt bardzo do przodu, z powodu chaotycznej natury procesu powstawania gier.



Rys. 2 Przykładowy **uproszczony** proces produkcji gry

Jeśli wszystkie kamienie milowe są osiągane o czasie, mamy (niemalże) gwarancję że końcowy produkt również będzie na czas. Jeśli choć jeden kamień milowy spóźnia się, nie zrobiono wszystkiego co było założone, **natychmiast** wiemy że z projektem dzieje się coś nie tak. Ustalając odpowiednio krótki czas pomiędzy kolejnymi kamieniami milowymi (np. 4-5 tygodni, jak to się zaleca w Agile), mamy szansę odpowiednio zareagować **już teraz.**

Warunkiem koniecznym (ale nie wystarczającym) aby opisany powyżej proces działał, jest wyznaczenie solidnych, realistycznych czasów wykonania zadań. Tylko bardzo doświadczeni programiści są w stanie regularnie takich dostarczyć. Ktokolwiek inny będzie srogo się mylił. Najczęściej popełnianym błędem jest niedocenienie zadań - słyszymy pewne siebie „*eee tam, to jest proste, zajmie mi to godzinkę, może dwie*" - po czym widzimy kodera powoli tracącego nerwy gdy zajmuje mu to dzień lub tydzień (może dwa).

2.2 Komunikacja w zespole

Dzisiejsze gry retail pisane są przez zespoły złożone z

kilkudziesięciu, w porywach więcej niż stu osób (nie wliczając outsourcingu). Przy takiej liczebności jest krytyczne dla powodzenia projektu, aby nie popełniać podstawowych błędów:

- **brak dublowania pracy**, gdy to samo zadanie wykonywane jest przez więcej niż jedną osobę
- wykonana **praca nie jest** potem **odrzucana** jako nieudana, lub już niepotrzebna
- **brak zatorów** (ang. bottlenecks) - gdy kilka osób regularnie czeka na wynik pracy jednej osoby, aby kontynuować ze swoimi zadaniami
- **usuwać zależności** (ang. dependencies) – ktoś czeka na koniec pracy innej osoby (różnicą w stosunku do zatorów jest brak regularności)
- **brak nieporozumień**, tworzone jest dokładnie to co potrzebne (ktoś powiedział X, inna osoba zrozumiała Y)

Uniknięcie tych błędów jest możliwe poprzez efektywną komunikację wewnątrz zespołu.

2.2.1 Komunikacja bezpośrednia

Podstawową cegiełką jest **spotkanie,** czyli zgromadzenie kilku osób i dyskusja mająca zakończyć się konkretnymi ustaleniami. Spotkania powinny być organizowane w przypadku wydarzeń i nietrywialnych problemów, które dotyczą większej ilości osób. Dobrze jeśli będzie stosowało się reguły z burzy mózgów: luźna atmosfera, każdy może się wypowiedzieć, nie ma głupich pytań. Przydatna jest osoba (z reguły nazywana facilitatorem) która będzie kontrolować przebiegu spotkania: by ktoś nie przejął dyskusji, by nie zeszła ona z tematu, by ludzie nie kłócili się itp. Facilitator w pewnym momencie musi powiedzieć "dość, teraz podejmujemy ostateczną decyzję". W końcu nie można spędzić całego dnia na czczej gadaninie.

Kolega który miał okazję pracować w kilku firmach w różnych rejonach kuli ziemskiej, zarzeka się, że organizowanie spotkań jest czymś, co wyróżnia firmy zagraniczne na plus od Polskich. Gdy pomyślę o projektach w których pracowałem w przeszłości, jestem w stanie przyznać mu rację. Niektórzy nawet twierdzą, że umiejętności komunikacji są najważniejszym co musi posiadać programista marzący o udanej *karierze*[4].

Spotkania znakomicie nadają się do przekazywanie wiedzy w obrębie zespołu, W Playlogic zorganizowaliśmy coś takiego jak Q&A, czyli Questions and Answers – godzinę raz w tygodniu poświęcamy na spotkanie wszyskich programistów znających UE3, którzy zadają i odpowiadają na pytania na które sami nie mogli / nie mieli czasu znaleźć odpowiedzi. Odpowiedzi zapisywane są na wewnętrznej wiki, aby w przyszłości dysponować podręczną, praktyczną skarbnicą wiedzy i swoistym FAQ.

Mimo, że kilkunastu programistów mogłoby w tym czasie siedzieć i pisać kod, rozwijając grę, jednak uznano że w dłuższej perpektywie (a w takiej należy planować jeśli do wydania gry pozostało jeszcze kilka miesięcy) pozwoli to polepszyć znajomość silnika, a co za tym idzie, efektywność koderów. Z nieco innych powodów, szef firmy ustalił **codziennie** rano spotkanie z programistami (swoisty standup ze Scrum) na którym zgłaszamy problemy, zależności, zadajemy ważne pytania. Podobne regularne spotkania organizują członkowie innych dyscyplin. Często robimy multidyscyplinarne spotkania on-the-fly (*z marszu*), gdy nagle pojawia się trudny problem.

2.2.2 Prezentacje i warsztaty

Innym sposobem dzielenia się wiedzą są prezentacje. Playlogic jest moją pierwszą firmą w której dość regularnie programiści przygotowywują się na jakiś temat nie związany z aktualną grą czy tym co robią, i dokonują prezentacji z rzutnikiem. Powody tego są identyczne jak w przypadku Q&A.

Inny przykład to Design workshops – zorganizowaliśmy je jako pomoc dla game designera który dopiero dołączył do Fairytale Fights i sam miałby problemy z zaprojektowaniem wszystkiego na czas. Kierownictwo całkiem logicznie uznało, że aby tworzyć grę, trzeba wiedzieć jak ona będzie wyglądać, tymczasowo więc przerzucono wszystkich ludzi do projektowania gry.

Odbyły się trzy 1.5 godzinne sesje, w czasie których podzieleni zostaliśmy na 5 osobowe zespoły, dostaliśmy konkretne problemy projektowe o których mieliśmy dyskutować i zapisać wyniki naszych dyskusji na kartce. W wyniku zbiorowej współpracy znacznie szybciej ustalono design, który tylko zyskał dzięki “burzom mózgów”, dodatkowo wywołało to entuzjazm w zespole, pomogło go lepiej scalić i ludzie wreszcie wiedzieli jaką grę przyszło im tworzyć. Same zalety.

2.2.3 Komunikacja na odległość

Aktualnie preferowanym sposobem komunikacji nie-bezpośredniej jest internet / intranet. Do tej pory, w komunikacji wewnątrz zespołu (jeden do jednego i broadcasting), sprawdziły się:

- Outlook
- forum
- mailinglista
- inne

Koniecznie muszę zaznaczyć że Outlook != Outlook Express i Outlook ! = Thunderbird. Mimo powierzchownych podobieństw, jakim jest obsługa poczty, są to inne programy. Outlook wspomaga pracę zespołową poprzez możliwości ustalania spotkań grupowych z automatycznym dodawaniem ich do kalendarza, przypominania o nich przez komunikaty („Za 5 minut zaczyna się twoje spotkanie w pokoju X"), współdzielenia zadań, notatek, predefiniowanych list kontaktów wewnątrz organizacji. Przyspiesza i ułatwia to wykonywanie tych nudnych, zajmujących czas (i pamięć) czynności. A co najważniejsze, jest bardzo prosty w obsłudze, wręcz idioto-odporny, oraz można na nim polegać, jak na mało którym programie Microsoftu.

Forum internetowe (choćby PHPbb) ma swoje zalety – lepiej niż Outlook grupuje wiadomości w wątki, jest dostępne z każdego miejsca w świecie, umożliwia edycję tego co już wysłano. Wadą jest brak tego wszystkiego co opisalem w poprzednim akapicie.

Mailinglista to antyczna metoda komunikacji w grupach developerów rozsianych po całym świecie, polegająca na wysyłaniu emaili na jeden adres, z którego następnie są one automatycznie rozsyłane do wszystkich zapisanych. Mimo prostoty (a może dzięki niej) i braku tylu opcji co Outlook czy forum, sprawdza się w praktyce, co miałem okazję zaobserwować uczestnicząc w projekcie dość dużej gry RPG tworzonej przez ludzi rozsianych po całej Polsce [9]. Sprawdza się również w ogromnych projektach open-source, jak np. jądro Linuxa.

Pod punktem *Inne* mam na myśli oprogramowanie będące mieszanką cech z tych wymienonych, wykorzystujące nowe pomysły np. PresentlyApp[6] będące mieszanką Outlooka i grup dyskusyjnych, z czatem zintegrowanych w jednej platformie www. Trac spopularyzowany przez Assemblę integruje wiki, system kontroli wersji, przydzielanie zadań developerom i kamienie milowe. Zaliczyłbym tu także Skype i programy do szybkiej komunikacji tekstowej (Instant Messaging, np. Gadu Gadu).

2.2.4 Dokumentacja

Do przechowywania wiedzy w formie łatwo dostępnej i edytowalnej istnieją właściwie tylko dwa sposoby:

- wiki
- udostępnianie dokumentów (.doc/.odt/.pdf/itp) po LANie lub w repozytorium np. SVN

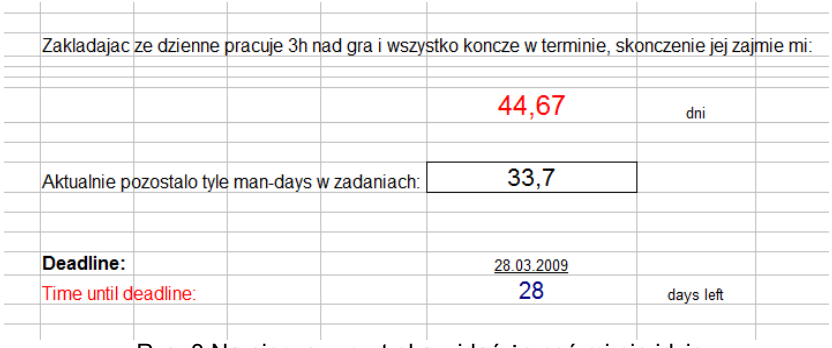
Wiki ma przewagę automatycznego wersjonowania i backupowania, dostęp z każdego miejsca (na świecie), łatwość linkowania i szukania dokładnie tego czego potrzeba. Dokumenty natomiast mogą być tworzone w rozbudowanym edytorze tekstów (Word/Open Office), nie ma więc problemów z zaawansowanym formatowaniem, zamieszczaniem tabel, dużych obrazków i drukowaniem.

Myślę że kardynalnym błędem byłoby ograniczanie się do jednego z powyższym narzędzi i robienie wszystkiego w nim na siłę. Zamiast tego lepiej dopasowywać narzędzia do zadań, używać ich do tego w czym są najlepsze.

Odnośnie zarządzania zadaniami przydzielonymi członkom zespołu i kontrolowania ich postępów, i pośrednio postępów całej gry: Microsoft Project często wymieniany w innych branżach jako podstawowe narzędzie managera... w gamedevie ludzie porównują używanie go do strzelania z rakietnicy do muchy i zalecają prostsze metody.

Można do tego wykorzystać Excela, lub jego odpowiednik. Każdy podsystem gry (np. rendering cieni, rendering trawy, wczytywanie poziomów, menu gry, obsługa kamery) otrzymuje osobny plik arkusza kalkulacyjnego. Zapisujemy w nich listę featursów do zaimplementowania z podziałem na zadania, i wyszczególnionymi takimi cechami jak łączna liczba dni do pełnej implementacji oraz ile już zostało zrobione. Główny arkusz kalkulacyjny projektu odwołuje się do nich, a po kliknięciu przycisku, makro powoduje automatyczne wczytanie danych z plików z zadaniami i odświeżenie ilości czasu do końca projektu.

Indywidualni koderzy posiadają osobiste arkusze, do których przenosimy zadania do wykonania w aktualnym sprincie. Arkusze za pomocą prostych formuł można przyjemnie zautomatyzować (przykład z Magic Pencil):



Rys. 3 Na pierwszy rzut oka widać że coś mi nie idzie...

2.4 Jasny podział na role w zespole

Co to daje? Ludzie nie tracą czasu zastanawiając się kto będzie wiedział jak rozwiązać ich problem. Decyzje są podejmowane przez osoby kwalifikowane do tego. Ograniczany jest chaos organizacyjny i różne przykre efekty tego stanu rzeczy.

Pierwsza sprawa to Szef. Boss. CEO. Project Manager. Producent.

Prezydent. Wódz. Mistrz. Wielki Szu - konkretna nazwa nie ma znaczenia, jest inna w każdej firmie, inna jest też jego rola i prerogatywy, jako że każda organizacja posiada unikalną strukturę i swoje naleciałości czasowe. Z tego powodu niemożliwe jest opisać kto czym powinien się zajmować w sposób ogólny, ale jedno jest pewne. Szef, to osoba z wysokimi umiejętnościami interpersonalnymi, posiadająca wizję oraz potrafiąca wymagać i motywować pracowników. Nawet nie będę udawał, że jestem w stanie tu opisać kompletnej charakterystki dobrego lidera, a co dopiero tego **jak** nim zostać. Zainteresowanych odsyłam do odpowiednich publikacji i szkoleń, które można znaleźć w bibliotekach i w internecie, np [8].

Szczerze mówiąc, na stanowisku szefa bardzo pasuje były doświadczony programista, mający wcześniej do czynienia z zarządzaniem ludźmi. Mało osób tak jak programista zna proces tworzenia gry od podszewki i posiada specjalistyczną wiedzę aby dogadać się z zespołem inżynierów, który w największym stopniu odpowiada za być albo nie być gry.

Potrzebny jest Project Manager z prawdziwego zdarzenia – sam główny programista to za mało, jego rola jest inna niż PMa, lepiej też dać mu więcej czasu na zajmowanie się kodem i zarządzanie programistami. Project Manager często zajmuje się robotą papierkową i szukaniem sposobów lepszej organizacji czasu/pracy.

Inne ważne role do obsadzenia: Game Designer i producent. Niekoniecznie muszą być obsadzone przez różne osoby np. Game Designer i producent, czy producent i szef studia ładnie się łączą. W przypadku bardzo dużych zespołów, lepiej jednak pozostawić te funkcje różnym osobm. Ważne jest aby dla wszystkich było jasne kto jest kim, za co odpowiada, i do kogo należy iść w razie kłopotow.

To zupełna oczywistość, ale nie mogłem się powstrzymać: programiści nie mogą zajmować się obsługą sieci, instalacją/ naprawami systemów, oprogramowania i sprzętu, pilnowaniem bezpieczeństwa sieci itp. – a taki przypadek widziałem na własne oczy (a nawet gorzej, bo sam takie rzeczy robiłem) - gdyż odciąga ich to od tego za im się płaci i na czym naprawdę się znają. Potrzebny jest zatem dedykowany administrator, niekoniecznie na pełny etat.

Każda dyscyplina reprezentowana przez co najmniej kilka osób powinna posiadać swojego przedstawiciela (tzw. *lead'a*), którym rzecz jasna powinna być osoba najbardziej doświadczona. Np. główny programista, dyrektor artystyczny, główny animator, główny projektant poziomów... To nie jest zabawa w armię czy “kto ma więcej tytułów”, albo działanie na zasadzie d**ziel i rządź**.

Zadaniem takiej osoby jest pilnowanie swoich podopiecznych, ustalanie, koordynowanie i pilnowanie wykonania zadań, upewnianie się że

posiadają wszystko co potrzebne aby wykonać swoje zadania, dwustronna komunikacja między wyższym szczeblem zarządzania a zespołem, pomoc zalegającym z terminami oraz wiele innych.

W efekcie tych wszystkich starań powinniśmy uzyskać klarowny podział zespołu, w którym decyzje podejmowane na górze hierarchii są efektywnie przekazywane i wykonywane na dole, zaś wszelkie problemy i pytania powstające na dole, są przekazywane do góry. Osiągnięcie tego drugiego jest trudniejsze i musi być szczególnie wspomagane przez kadrę zarządzającą. Tyle teoria i ideał.

2.6 Metodyki inżynierii oprogramowania

Waterfall (klasyczny model tworzenia projektów) nie sprawdza w branży gier. Prawdziwe wymagania na początku projektu są nieznane, zmieniają się z czasem i zależą w dużej mierze od tego co się okaże w trakcie implementacji. Nowe narzędzia i technologie sprawiają, że trudno przewidzieć czas implementacji i koszty ich utrzymywania. Inne powody podał Steve McConnel w biblii programistów, Code Complete[5].

Stale poszukuje się lepszych metodyk produkcji oprogramowania. Aktualnie popularne jest bycie *zwinnym* – stosowanie metodyk z rodziny Agile. Wydaje mi się, że w niewielkim stopniu ta popularność jest przejawem mody, które niekiedy się zdarzają także w inżynierii oprogramowania (Java?), lecz wynika ze zbiorowego doświadczenia, eksperymentowania i wyciągania wniosków z porażek i sukcesów na przestrzeni lat.

Lekkie/zręczne metodyki skupiają się na posiadaniu ciągle działającej wersji, ciągłej integracji, krótkich iteracjach (do kilku tygodni), samoorganizacji, oraz jeszcze innych. Agile to skuteczny, sprawdzony sposób dostarczania kompletnego produktu na czas i budżet. Praktyki te szczególnie świetnie pasują do teamów developerskich gier komputerowych.

Scrum[3] proponuje częste kilkutygodniowe iteracje zwane *sprintami,* które mają dostarczyć działającego i przetestowanego zestawu nowych funkcjonalności; codzienne 15 minutowe spotkania programistów – *standupy;* przechowywanie listy zadań do wykonania w *backlog*u. W czasie sprintu lista zadań jest niezmienna. Przedstawiciele klienta blisko współpracują z zespołem aby pomóc mu zrozumieć wymagania i ustalać priorytety. Po wdrożeniu Scrumu w procesy tworzenia oprogramowania w setkach działających przedsiębiorstw, zaobserwowano średnio dwukrotne zwiększenie wydajności, w najlepszych przypadkach nawet czterokrotne[13].

Programowanie ekstremalne zaproponowało ciekawą koncepcję programowania parami. Najpewniej nie jestem jedynym, który zauważył że

rozwiązywanie problemów z drugą osobą przy boku, przychodzi znacznie łatwiej, przy czym jest to wzrost ekspotencjalny, a nie zwykły liniowy. *Unit testy* przydają się szczególnie do testowania niewielkich kawałków kodu o przewidywalnym wejściu i wyjściu które łatwo można porównać, jak np. funkcje matematyczne, jednak ich zastosowanie w gamedevie jest ograniczone jako że nie da się napisać unit-testa do zdecydowanej większości featursów w grze. Kolejną polecaną praktyką w XP jest *ciągła integracja*, czyli zestaw praktyk pomagających uniknąć *integration hell*, gdy zmiany wprowadzone w lokalnej kopii repozytorium, są trudne do połączenia z najnowszą wersją na serwerze.

Wymieniłem dwie popularne metodyki ale to wierzchołek góry lodowej. Podobnie jak w innych przypadkach, odradzam bezrefleksyjne kopiowanie wszystkiego z jednej metodyki. Wydaje mi się, że wybieranie z nich tych elementów które pasują do naszej organizacji powinno być bardziej praktyczne, choć jednocześnie będzie trudniejsze.

2.6.1 Systemy zarządzania kodem / assetami, wspomagające pracę

To jest jak najbardziej podstawowa i oczywista sprawa, więc będzie krótko. W branży wykorzystywanych jest głównie kilka popularnych, sprawdzonych systemów wersjonowania danych. W przypadku kodu będzie to open-sourcowy Subversion (następca CVS), z popularnym klientem TortoiseSVN, lub integrującym się z Visual Studio AnkhSVN. Alternatywą może być komercyjny Perforce, który odchodzi trochę od modelu działania CVS/SVN, stawiając m.in. na wydajność.

Ogromne ilości assetów (zasobów) potrzebnych w dzisiejszych grach next-genowych możemy przechowywać w normalnym repozytorium SVN (który wspiera diffy plików binarnych) i wiele zespołów tak robi. Możemy też użyć do tego wyspecjalizowanego narzędzia jakim jest komercyjny i drogi Alienbrain, jednak potencjalne zyski z użycia go w dużym zespole są (podobno) warte każdych kosztów.

Warto również poświęcić czas na przygotowanie szybkiego **zautomatyzowanego** build serwera, budującego cały projekt od podstaw aż do gotowego instalatora, tworzącego backupy, wysyłającego maile z powiadomieniami o problemach itd. Programiści C++ będą ubóstwiać firmę w której używa się systemu rozproszonej budowy kodu, dystrubuującego kompilację na nieużywane aktywnie maszyny (np. Incredibuild czy distcc); będą kochać firrmę która zapewnia narzędzia do efektywnej nawigacji w kodzie jak Visual Assist X czy Resharper. Wszystko to dla skrócenia czasu iteracji *kodowanie-kompilacja-uruchomienie-poprawki*, gdyż przekłada się to szybszy developing, mniejsze rozproszenie uwagi, dłuższe przebywanie programistow w tzw. *zone* (stan podwyższonej aktywności mentalnej i całkowitego skupienia) ---> lepszą grę i realne zyski przedsiębiorstwa.

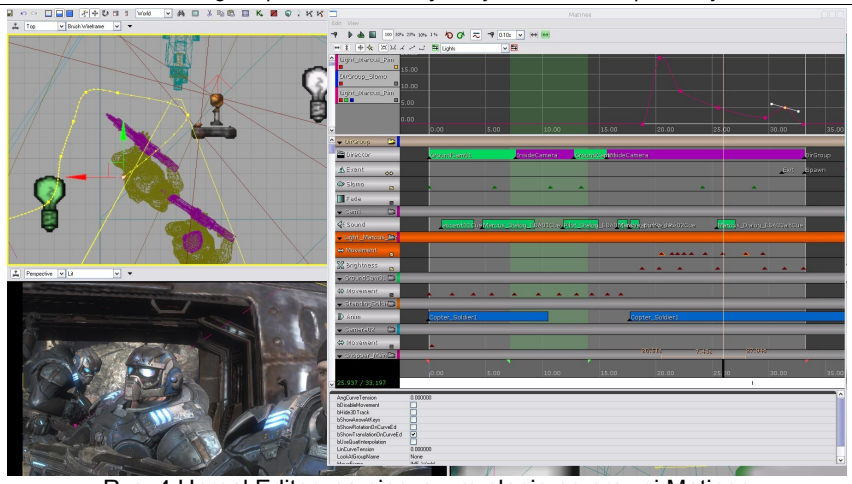
2.7 Gry sterowane danymi

Gra sterowana danymi, to taka w której różnego rodzaju stałe i reguły normalnie "zaszyte" głęboko w kodzie, są wczytywane z plików konfiguracyjnych i zasobów ulokowanych na dysku twardym. Te pliki mogą być dowolnie proste lub skomplikowane, w tym drugim przypadku lepiej jednak napisać narzędzia do ich graficznej edycji. Ułatwia to znakomicie prototypowanie wczesnych wersji, oraz iteracyjne tworzenie właściwej gry później, szczególnie faza końcowa (*polishing, tweaking*) wiele zyskuje. To jest klasyczna praktyka programistyczna, którą można rozszerzyć na inne aspekty.

Kolejnym poziomem będą języki skryptowe udostępniające możliwość tworzenia **logiki** gry bez modyfikowania właściwego kodu. Uniezależnia to w pewnym stopniu game designerów od programistów. Dodatkowo zyskujemy automatyczne wsparcie dla modów tworzonych przez graczy, co jest bardzo popularne. Minusem jest trudniejsze debugowanie i konieczność uczenia designerów pisania skryptów.

Na własnej skórze odczułem zalety języków skryptowych po zaadaptowaniu AngelScript[9] do silnika Magic Pencil, gry którą aktualnie tworzę. W ciągu jednej nocy otworzyły się możliwości o których nie śmiałbym wcześniej marzyć. Było tak mimo tego, że Magic Pencil wykorzystywał dość mocno data-driven-approach poprzez rozbudowane pliki konfiguracyjne obsługiwane przez open-sourcową bibilotekę SDL_Config[10], również mojego autorstwa. Ale prawdopodobnie najlepszym przykładem podejścia całkowicie data-driven jest Unreal Engine 3.

Unreal Engine 3 jest najbardziej popularnym, najczęściej licencjonowanym silnikiem na PC i konsole aktualnej generacji (zwyczajowe określenie next gen), trzykrotnie pod rząd laureatem wyróżnienia magazynu GDmag na technologię popychającą branżę do przodu. W znacznym stopniu przyczyniły się do tego jego rozbudowane, wydajne, ale także proste w użyciu narzędzia. Wizualny język "skryptowy" Kismet, edytor "filmów i ścieżek" Matinee, edytor efektów cząsteczkowych Cascade, edytor materiałów (shaderów) i wiele wiele innych zostały zgromadzone pod skrzydłami jednego głównego programu – Unreal Editor, umożliwiając nie-programistom kompletną edycję WYSIWYG konfiguracji gry, poziomów, zachowań postaci, menu, efektów specjalnych, itp. O podobnych tematach będę się wypowiadał w punktach 3.1 i 3.2.



Rys. 4 Unreal Editor, na pierwszym planie po prawej Matinee

3. Bad... things to avoid

3.1 Zależnosci

Zależności (dependencies) trzeba wpierw rozpoznać. W praktyce sprowadza to się do pilnowanie aby część zespołu nie siedziała bezczynnie, czekając na efekt prac drugiej części zespołu, gdyż opóźnienia projektu i straty finansowe wynikające z tego są niewyobrażalne.

Największym bottleneckiem (wąskim gardłem) zazwyczaj są programiści, na których barkach spoczywa tworzenie: silnika gry, właściwej gry, narzędzi do gry, usuwanie bugów itd. a do tego na bieżąco wspieranie innych dyscyplin. Im mniej inni ludzie będą musieli zawracać głowę programistom aby coś zrobić, tym bardziej zwiększa się wydajność obu grup, dlatego każdy pomysł jak odciążyć programistów jest na wagę złota.

Zazwyczaj przenosi się zachowanie gry z kodu, na zewnątrz do plików edytowanych wizualnymi edytorów i łatwego dostępu, ale o tym rozpisywałem się już w punkcie 2.7.

Bywa też na odwrót – programiści mogą być zależni np. od animatorów, czekając na animacje do modeli. Rozwiązaniem na krótką metę jest stosowanie modeli tymczasowych (*mockupów*), których się używa gdyż „czegoś użyć trzeba”, a które zostaną zamienione na właściwe, gdy tylko te będą dostępne. Niestety, nie zawsze to jest możliwe lub ma sens, dlatego lepiej przewidzieć takie problemy dużo wcześniej i ustalić priorytety.

3.2 Nie-efektywne narzędzia

Kierownik projektu ma do dyspozycji generalnie 2 czynniki: **ludzi** i **pieniądze**. Są one wymienialne ze sobą w różnych proporcjach, ale zawsze dąży się do optymalizacji ich użycia. Do tego dochodzi jeszcze zarządzanie/ walka z z czasem. Powszechnie wiadomo że czas to pieniądz. Więc kiedy pojawia się nowa potrzeba wykonywania nie-trywialnego zadania w sposób efektwny, generalnie do wyboru jest: kupno produktu (*middleware?*) od innej firmy, przydzielenie ludzi aby stworzyli taki produkt do użytku wewnętrznego, albo znalezienie darmowego/open sourcowego odpowiednika. Wbrew pozorom, wielokrotnie najlepsze dla firmy chcącej **naprawdę** oszczędzać pieniądze, będzie kupno istniejącej, dopracowanej technologii.

Jako mini anegdotkę pokazującą działanie tej zasady w praktyce, przytoczę sytuację, gdy w Playlogicu dyskutowaliśmy o tym jaki wybrać program do instalacji Fairytale Fights: komercyjny za 10 000$ oferujący wszystko czego potrzebowaliśmy out-of-the-box, czy darmowy open-sourcowy, ale który ktoś z nas musiałby poznać i ulepszyć, aby zawierał to czego potrzebujemy (poświęcając na to czas który w innym wypadku przeznaczyłby na ulepszanie własciwej gry). Na sali siedziało 10 programistów i kilka osób z kadry zarządzającej, czyli zawodów zarabiających najwięcej w branży gier. Po 15 minutuach ktoś zauważył „*ale o czym mu tu właściwe gadamy, całe to spotkanie kosztowało nas już więcej niż te 10 tysięcy, bierzemy tego komercyjnego*”. Jak już napisałem, czas to pieniądz, a czas game developerów jest szczególnie *cenny*.

3.3 Bugi

Generalnie, jeśli firma w której pracujesz nie nazywa się Blizzard, nie istnieje sposób na całkowite ich uniknięcie w ostatecznej wersji programu. Zaś w czasie produkcji będzie ich całe mnóstwo, dlatego trzeba umieć z nimi walczyć. Istnieją dwa, uzupełniające się podejścia: walka z przyczynami, albo ze skutkami.

Przyczyną problemu są programiści, którzy piszą kod zawierający błędy. Logiczne jest więc, że im lepszy jakościowo kod zostanie napisany, tym mniej będzie błędów. Jaki kod jest wysokiej jakości? To zaskakująco głębokie pytanie, odpowiedź nie jest prosta i nie podejmuję się jej udzielenia. Istnieje natomiast dodatnia korelacja pomiędzy doświadczeniem programisty, a jakością pisanego kodu (ale słabsza niż by się wydawało). Tak więc, częściowym sposobem na tworzenie software z jak najmniejszą ilością błędów, jest zatrudnianie najlepszych programistów na jakich nas stać i jakich możemy znaleźć, zachęcanie ich do koncentrowania się na pisaniu dobrego, czystego, ładnego, przenośnego, elastycznego,

utrzymywalnego (*maintainable*) itp. itd. kodu.

Inne sposoby z dziedziny zarządzania, na radzenie sobie z przyczyną błędów w grach, zostały szczegółowo opisane wcześniej: nowoczesne metodyki programowania (jak Agile), porządne zaplanowanie struktury projektu z określeniem wymagań i terminów na czele, używanie efektywnych narzędzi itd.

Sposobem na walkę ze skutkami, są rygorystyczne testy / QA (Quality Assurance) i posiadania **bugtracka** czyli narzędzia/bazy danych do przechowywania listy bugów i problemów (issues). Obowiązkiem testerów przy zgłaszaniu błędu jest wypełnianie pól takich jak numer wersji gry, dokładny sposób reprodukcji problemu, imię osoby która go znalazła (do kontaktu), niekiedy załączenie obrazka lub filmu, czasami też przypisanie osoby która nim się zajmie. Popularne bugtracki to Bugzilla, Mantis, Trac, OnTime, Flyspray, FogBugz... i Excel.

3.4 Prolonged crunch

Ta branża ma to do siebie, że ludzie pracują często dla przyjemności i możliwości kreatywnego wyrażenia się poprzez tworzenie; zarabiane pieniądze są ważne, ale nie na pierwszym planie. Satysfakcja z tworzenia gier jest na tyle duża, że pracownicy sami z siebie zostają po godzinach, pracując za darmo, tylko po to aby ICH gra, nad którą spędzili ostatnie lata swojego życia, była najlepsza na świecie.

W takiej sytuacji nikogo nie dziwi, że w większości firm **nie** płaci się za nadgodziny. Co więcej, w wielu z nich wymyślono coś takiego jak "niepłatne obowiązkowe nadgodziny", gdy od pracowników oczekuje się takiego właśnie zachowania, a co jeszcze bardziej śmieszne, pracownicy wcale się temu nie dziwią i akceptują to jako normalne. Zjawisko to nazywa się *crunch mode* i występuje na całym świecie, od Tokio, przez Kraków, aż do Nowego Yorku.

Niewielki crunch na koniec jest akceptowalny i niemal nie do uniknięcia. Podobne zachowania występują we wszystkich przedsięwzięciach człowieka, od tworzenia oprogramowania, przez inżynierię lądową (Euro2012?), po sesję na studiach. Ale chyba nikt nie lubi regularnie spędzać 16+ godzin na dobę przed komputerem, z czego połowę za darmo.

Jest to szczególnie prawdziwe w krajach zachodnich, gdzie średni wiek pracowników gamedevu już dawno minął 30 lat. Nie są już młodzi, głupi, naiwni... o przepraszam, entuzjastycznie nastawieni. Mają rodziny, przyjaciół, życie poza firmą itd. IGDA [14] od lat stara się o poprawę jakości życia ludzi pracujących w branży gier.

4. Ugly... things you have no control over (or you think you have)

4.1 Czynnik ludzki

Ludzie produkują oprogramowanie. Od ich wkładu zależy jakość produktu. Zwykli rzemieślnicy nie są źli, ale najlepsze efekty powstają gdy profesjonaliści wkładają w pracę także swoje serce, gdy naprawdę im zależy. Widać to szczególnie w dbałości o szczegóły. Gry tworzone z sercem mają większą szansę na przeżycie na dyskach graczy, dłużej więc będzie się o nich mówiło, co ma z kolei duży wpływ na ewentualne sequele.

To czy ludziom zależy, jest funkcją nieskończonej liczby czynników, wśród których na pewno znajdą się: czy lubią projekt i swoje zadania, czy są wolni artystycznie czy może kontrolowani i ograniczani na każdym kroku, czy dobrze zarabiają, czy lubią tych z którymi przyszło im pracować, czy odpowiada im atmosfera w pracy. Istnieje nawet ciekawe prawo Conwaya[11], twierdzące że:

> *Any organization which designs a system... will inevitably produce a design whoose structure is a copy of the organization's communication structure.*
>
> Melvin Conway

W [12] wynikła dyskusja jak to prawo manifestuje się na przykładzie Powerpointa i Microsoftu.

Powiedziałem już, że od tego co się dzieje w zespole, od relacji międzyludzkich, zależy czy gra będzie typową rzemieślniczą robotą za pieniądze/ dla pieniędzy, czy czymś więcej. Możnaby przypuszczać, im atmosfera w zespole jest lepsza – tym gra będzie lepsza.

Niestety, w każdej organizacji pojawiają się konflikty. Rozwiązywanie ich zależy od przypadku do przypadku. Niektórzy ludzie nie są ze sobą kompatybilni. Niektórzy są chorobliwie nieśmiali (szczególnie częste wśród programistów w gamedevie), inni są ich przeciwieństwem. Wszyscy mamy swoje przyzwyczajenia, które niekoniecznie muszą być lubiane przez innych, a każda drobnostka może zostać rozdmuchana do nieproporcjonalnych rozmiarów.

W drastycznych przypadkach jedynym sensownym rozwiązaniem może być zwolnienie pracownika/pracowników którzy sprawiają problemy... ale lepszym pomysłem jest zapobieganie chorobie zamiast leczenia. Niemożliwe jest zmuszenie ludzi do polubienia siebie, ale zauważono, że w przypadku mężczyzn (którzy stanowia zdecydowaną większość siły roboczej

gamedevu), grupowe działania zawiązują "braterstwo krwii". Jeżeli fakt tworzenia wspólnie gry nie wywołuje takich reakcji, być może warto zastanowić się nad... imprezami integracyjnymi? W Playlogic taką rolę w mini-skali spełniają Q&A programistów.

4.1.1 Wydawca

Wydawca też człowiek. Kyle Gabler z 2d boy powiedział, że niemożliwa jest relacja bez napięć, między studiem gamedev a wydawcą. Co więcej, powiedział nawet iż byłaby to niechciana sytuacja, gdyż mogłaby świadczyć o tym, że stronom nie zależy na powodzeniu projektu, jako że w takiej sytuacji oczekiwania odnośnie drugiej strony są coraz wyższe. Napięcia mogą pojawiać się nawet w jednej firmie, która pod swoimi skrzydłami posiada własne studio i odział wydający gry na rynku; niestety przykładu nie wolno mi podać.

Niemożliwe jest w tak krótkim referacie napisać coś więcej na tak rozległy i trudny temat jak **człowiek**. Dobre sposoby na radzenie sobie z ludźmi opisano w [8] i [10].

4.2 Konkurencja

Pod koniec roku 2008 nastąpił prawdziwy wysyp długo oczekiwanych gier, potencjalnych hitów: Fallout 3, Far Cry 2, Mirror's Edge, Dead Space, Little Big Planet, Gears of War 2, Fable 2, Call of Duty - to przykłady dopracowanych, porządnych gier wydanych w okresie od listopada do grudnia. Takie nagromadzenie wspaniałych gier na przestrzeni 2 miesięcy wywołało rozterki u graczy, którzy również z powodu rozwijającej się recesji, musieli ostro selekcjonować co kupują. Z reguły wybierali to co naj naj naj (Fallout 3 i Gears of War 2 w tym przypadku), i niestety trochę słabsze, ale w dalszym ciągu mocno wyróżniające się gry, sprzedały się poniżej oczekiwań, przynosząc poważne straty.

Zachowanie się wydawców jest logiczne: starają się wydać gry w okolicach świąt Bożego Narodzenia, gdyż wtedy konsumenci ruszają do masowych zakupów. W branży gier to zachowanie jest obserwowane od ponad 20 lat. W tym roku jednak było błędem. Możemy się spodziewać, że w 2009 wydawcy nie popełnią tego błędu, umieszczając hity także w innych okresach niż święta... ale czy na pewno?

5. Wnioski

Stworzenie wspaniałej gry to niezwykle wymagające zajęcie. Stworzenie takiej gry w dużym zespole, przy dzisiejszej (zażartej) konkurencji, wysokich wymaganiach graczy, i jeszcze zarobienie na niej wystarczająco, aby projekt się spłacił i zostało coś na następne... to nie lada wyzwanie.

Powyżej opisałem praktyczne porady i sposoby na uniknięcie różnych podstawowych błędów w **Twoich** projektach. Nie jest to pełny albo nawet choćby częściowo skończony sposób na napisanie udanej gry – taki nie istnieje. Rokrocznie uświadczamy zbyt wielu porażek znanych, doświadczonych teamów developerskich, zbyt wielu sukcesów nieznanych wcześniej zespołów, by ktokolwiek miał nadzieję, że **sztukę** tworzenia gier uda się spisać na kartce papieru w formie podpunktów „Sprawdzony sposób na tworzenie udanych gier 1.0" i z powodzeniem stosować. Myślę jednak, że przyswojenie wiedzy z tego referatu jest lepsze niż zignorowanie go i wymyślanie własnych praktyk ad hoc.

There is no silver bullet, jak kiedyś powiedział ktoś znany[1], w innym kontekście. Najważniejsze to mieć otwarty umysł i obserwować z uwagą co się dzieje z projektem. Mam nadzieję że w swoim referacie pomogłem Tobie uporządkować myśli i dostarczyłem tematów do rozmyśleń.

Bibliografia

[0] Gamasutra.com – portal dla profesjonalnych game developerów o technicznej stronie tworzenia gier, Kotaku.com – znany projesjonalny blog o grach i tym co piszczy w branży
[1] Mityczny Osobomiesiąc, Fred Brooks
[2] Cowboy coding, np. http://c2.com/cgi/wiki?CowboyCoding
[3] Agile Software Development with SCRUM, Ken Schwaber, Mike Beedle
[4] http://weblog.raganwald.com/2008/04/single-most-important-thing-you-must-do.html
[5] Code Complete, Steve McConnel
[6] www.presentlyapp.com

[7] www.angelcode.com/angelscript/
[8] wszystko Anthony'ego Robbinsa
[9] www.wladca.pl
[10] How to win friends and influence people, Dale Carnegie
[11] http://en.wikipedia.org/wiki/Conway's_Law
[12] http://www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=00025o&topic_id=1
[13] http://video.google.com/videoplay?docid=8795214308797356840
[14] International Game Developers Association, http://www.igda.org/qol/

Good, bad and ugly experiences, and lessons learned on game software engineering

Abstract

In my article I describe good and bad practices in gamedev / software developement I learned while working on several, vastly different projects, problems that might arise and ways to solve them.