

目次

第一章	序論.....	4
1.1	本章の構成	1
1.2	研究背景	1
1.3	研究目的	1
1.4	研究方法	1
1.5	プロジェクトマネジメントとの関連.....	2
1.6	本論文の構成.....	2
第二章	Minecraft と MOD について	1
2.1	本章の構成	3
2.2	用語説明	3
2.3	Minecraft とは.....	3
2.4	MOD とは	4
2.5	MOD の導入方法	7
第三章	GitHub について	3
3.1	本章の構成	10
3.2	バージョン管理システムについて.....	10
3.2.1	集中型バージョン管理システムについて	10
3.2.2	分散型バージョン管理システム.....	11
3.3	Git について	12
3.3.1	Git の特徴	13
3.4	GitHub とは	13
3.4.1	GitHub の基本用語.....	14
3.4.2	Git コマンド	16
第四章	調査.....	10
4.1	本章の構成	31
4.2	調査環境・調査方法.....	31
4.2.1	調査環境解説.....	31
4.2.2	Ubuntu の特徴.....	31
4.3	調査環境構築	34
4.4	調査方法	36

4.5 調査結果	43
4.5.1 Google Trends.....	45
4.5.2 ニコニコ動画.....	47
4.5.3 仮説	48
第五章 考察.....	31
5.1 考察	51
5.1.1 ライフサイクル.....	57
5.2 研究の反省・まとめ.....	60
5.3 謝辞	60
図 1 Minecraft ゲームフロー	4
図 2 影 MOD 使用例	5
図 3 デフォルトゲーム画面.....	6
図 4 テクスチャ&スキン変更 MOD 使用例.....	6
図 5 Minecraft Forge 画面	7
図 6 インストーラー画面.....	8
図 7 mods フォルダ追加後.....	8
図 8 集中型バージョン管理システム.....	11
図 9 分散型バージョン管理システム.....	12
図 10 ブランチの git rebase 処理前	21
図 11 ブランチの git rebase 処理後	22
図 12 プログラム概要図.....	42
図 13 Minecraft の MOD 数.....	43
図 14 Minecraft の売上げデータ.....	44
図 15 ubuntu 画面	45
図 16 Google Trends の検索傾向	46
図 17 Google Trends の人気検索ワード.....	46
図 18 仮説.....	48
図 19 ニコニコ動画内の Minecraft に関する動画数.....	50
図 20 Minecraft と MOD のグラフ	51
図 21 ライフサイクル.....	52
図 22 目的に対する達成条件.....	52
図 23 Minecraft の売上と MOD プロジェクト数の増加幅の比較.....	53
図 24 動画数と Minecraft の売上の増加幅比較.....	54
図 25 新しいライフサイクルの売上.....	58
図 26 従来のライフサイクルにおける売上.....	58

図 27 従来のライフサイクル図.....	59
図 28Minecraft のライフサイクル図.....	59
表 1 gunplot 解説.....	41
表 2 コマンド解説.....	41
表 3 歴代 PS3 売上上位 10 作品	エラー! ブックマークが定義されていません。
表 4 歴代 Wii 売上上位 10 作品.....	エラー! ブックマークが定義されていません。
表 5 歴代 Wii U 売上上位.....	エラー! ブックマークが定義されていません。

第一章 序論

1.1 本章の構成

第一章では、本論文の序論を述べる。研究背景、研究目的、研究方法、プロジェクトマネジメントの関連、本論文について記述する。

1.2 研究背景

ゲームには **MOD (Modification)** あるいは **Add-on** と呼ばれる拡張プログラムが存在する。**MOD** はユーザーが独自に開発しているため、基本的には無料で配布される。**MOD** によって追加された要素がユーザーに認められ、評価されることにより、ゲーム自体の知名度が上がり、その売上も上がるのが期待されている。その反面、制作側の意図しないゲームの動きが可能となってしまう。そのため制作側は **MOD** の導入を許可していないことの方が多い。知名度が上がるというメリットもあるが、制作側の意図していない事象が起きてしまうことのデメリットの方が大きいと考えてのことだろう。しかし、**Minecraft** という商業用 PC ゲームは **MOD** の導入を許可し、尚且つ売上を伸ばしている数少ないゲームである。

Minecraft とはマルクス・ペルソンとその会社 (**Mojang AB**) の社員が開発したサンドボックスゲームである。このゲームの特徴は、立方体のブロックで世界が構築されており、ブロックを設置したり破壊したりすることができることと、**Crafting** (工作) という機能が存在し、ブロックやアイテムを組み合わせる新しいアイテムを作ることのできる 2 種類の大きな特徴がある。サンドボックスゲームは多々存在するが、**Crafting** という機能があるのは **Minecraft** だけである。ゲーム自体はシンプルであるがために **MOD** による拡張でゲーム内での選択肢が増加し、プレイヤーの自由度が高い為、**MOD** との相性はとても良い。ここでは **MOD** の導入を許可した商業用ゲームの 1 つの成功事例として **Minecraft** を研究対象とする。

1.3 研究目的

本研究は、ゲームビジネスにおける **MOD** の影響について研究する。**Minecraft** は、その販売数が約 1700 万本 (2014 年 8 月時点) の大ヒットゲームであるが、この成功には、**MOD** を許可してボランティアの開発者を取り込み、ゲームを改善させて知名度を上げるという戦略が寄与していると思われる。

この仮説を検証することを本研究の目的とする。

1.4 研究方法

GitHub というホスティングサービスでは、**Minecraft MOD** の開発プロジェクトが行われ

ている。そのため、GitHub 上で行われている Minecraft MOD プロジェクト数を月単位で集計し、その変化を見ることができる。MOD プロジェクトのデータは GitHub から抽出し、そのプロジェクトがいつ開始されたのかを調査する。過去に GitHub 上で行われているプロジェクトの時系列データを抽出する研究が行われていたため、それを参考に Minecraft MOD プロジェクトの時系列データの抽出が可能であると考え。さらに、Minecraft の売上を月単位の累積グラフにして変化を見る。2 種類のグラフが同じ様な右上がりのグラフになった場合 MOD プロジェクト数の変化が売上の変化に影響を与えていることになる。

1.5 プロジェクトマネジメントとの関連

ゲーム制作のマネジメントにおいて、MOD との関係性はゲーム制作プロジェクトの戦略を立てるために必要な情報であると考え、本研究は戦略マネジメントの分野に関連性があると言える。

1.6 本論文の構成

第一章では序論、第二章では研究対象である Minecraft と MOD について用語説明等を交え詳しく記述する。第三章ではバージョン管理システムについて、GitHub を中心として記述し、本調査のチケットのデータを抽出する対象である、GitHub についての解説、GitHub の API の説明を記述する。第四章ではプロジェクトマネジメントとの関係を記述する。第五章では具体的な調査方法、調査経過、データを示し、データ解析を行い、第六章で考察、まとめを行う。

参考文献

- [1] 大塚弘記. GitHub 実践入門 Pull Request による開発の変革. 株式会社技術評論社, 2014.
- [2] The GitHub Blog. The Octoverse in 2012. 2012-12-19.
<https://github.com/blog/1359-the-octoverse-in-2012> (参照 2014-09-17).
- [3] Gigazine. 「Microsoft が「Minecraft」開発元を 2680 億円で買収、その経緯とは？」, 2014-09-16. <http://gigazine.net/news/20140916-microsoft-acquired-minecraft/> (参照 2014-09-18).
- [4] 久保孝樹. チケットを活用するオープンソースソフトウェア開発の実態調査. 千葉工業大学, 2013, 卒業論文.

第二章

Minecraft と MOD について

2.1 本章の構成

本章では、本論文の調査対象である Minecraft の基礎知識、MOD の基礎知識を述べる。

2.2 用語説明

①サンドボックスゲーム…オープンワールドに近いジャンルで、「決まった目的が存在せずプレイヤーが自由に行動できる」「決まった攻略手順がないもの、選択肢の強要をされないもの」タイプのゲームのこと。

②テクスチャ…コンピュータグラフィックスにおいて、3次元オブジェクト表面に貼り付けられる模様。

③スキン…コンピュータのアプリケーションソフトウェアなどで、ユーザインタフェースの外観表示を変更できる機能。

④Add-on…アドオンまたはアドインとも呼ばれるソフトウェアに追加される拡張機能のこと。アドオンを追加するにはソフトウェアがアドオンの導入を前提として設計されている必要がある。

2.3 Minecraft とは

第一章で述べた通り Minecraft は、2009 年 5 月 10 日に Notch 氏(本名:Markus Persson)が開発を始めたサンドボックス型のものづくりゲームである。レトロゲーを想起させるドットテストのブロックが溢れる世界で、プレイヤーは建物やその他のものを自由に創造することが出来る。未開の土地を探索したり、洞窟を探検したり、モンスターと戦ったり、植物を育てたり、新しいブロックを手に入れ配置することで様々なものを作ることができる。想像力次第で小さな家から、ドット絵、地下基地、巨大な城まで何でも作ることが可能である。

さらに、マルチプレイが存在し、協力して巨大な建築物を作ることや、拡張プログラム次第で Player VS Player で他プレイヤーと戦うことも可能である。

Minecraft のゲームフローは以下である。

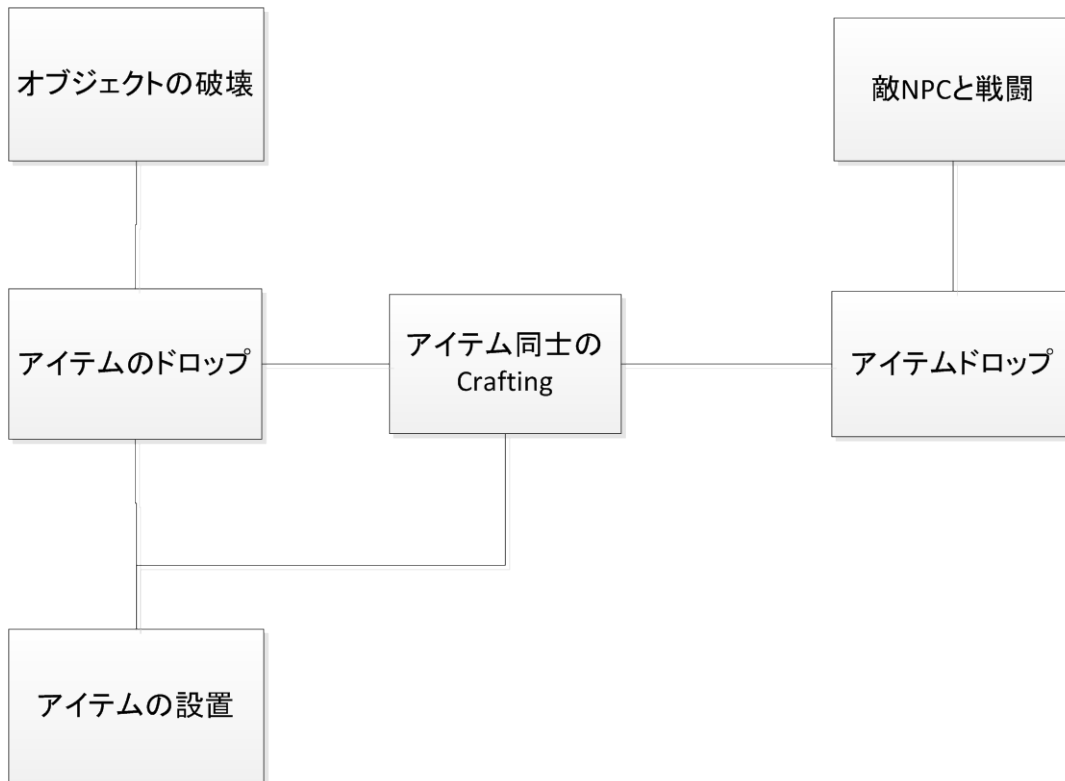


図 1 Minecraft ゲームフロー

複雑なゲームフローになればなるほど MOD による拡張機能の成果は見られず，ゲーム自体が単純なものであればあるほど拡張機能の成果は大きくなる．そのため，図のように単純なゲームフローとなっている Minecraft は，MOD による拡張機能の追加が容易で相性が良いと考えられる．このことから Minecraft 自体 MOD によるゲームの拡張を視野に入れて開発していたと推測することができる．その他の商業用ゲームは MOD によるゲームの拡張を許可せず，自分たちが意図したゲームをユーザーに提供している．この MOD に対する対応の違いが，本研究において Minecraft を調査対象とした最大の理由である．

2.4 MOD とは

こちらも第一章で述べた通り，MOD (Modification) あるいは Add-on と呼ばれる拡張プログラムである．MOD を導入することで本来のゲームのグラフィックや様々なデータを改造することができ，そのゲームのグラフィックエンジンや物理エンジンなどの基本システムを用いつつ，本編とは別のシナリオやグラフィック，モデル，システムで遊べるようになる．MOD の歴史はまだ浅く，90 年代半ば以降，欧米の FPS で MOD が大流行したのをきっかけに名前が知られ始める．この FPS では開発者が自社のゲームエンジンを使って同社のゲームを拡張することを許可し様々なバージョンを生み出していったことが評価された．

すなわち、MODの開発、導入を許可したことによって、結果的にはゲーム自体を発展させたことになる。上記にも述べた通り、MinecraftにおいてもMODの導入が許可されており、公式に発表するバージョンアップの他に、MODによる拡張でゲームの発展をユーザー自身でしていることになる。

実際にMinecraftで使われているMODの例を挙げる。

- 例1 Minecraft Forge…MODを導入するための前提MOD。主に他MODで拡張する際のベースとなるシステムを改造する。
- 例2 影MOD…グラフィックの変更用MOD。図2参照
- 例3 テクスチャ&スキン変更MOD…グラフィックの変更と共に画面表示を変更するMOD。図3、図4参照



図 2 影 MOD 使用例 図は考文[1]の引用元より

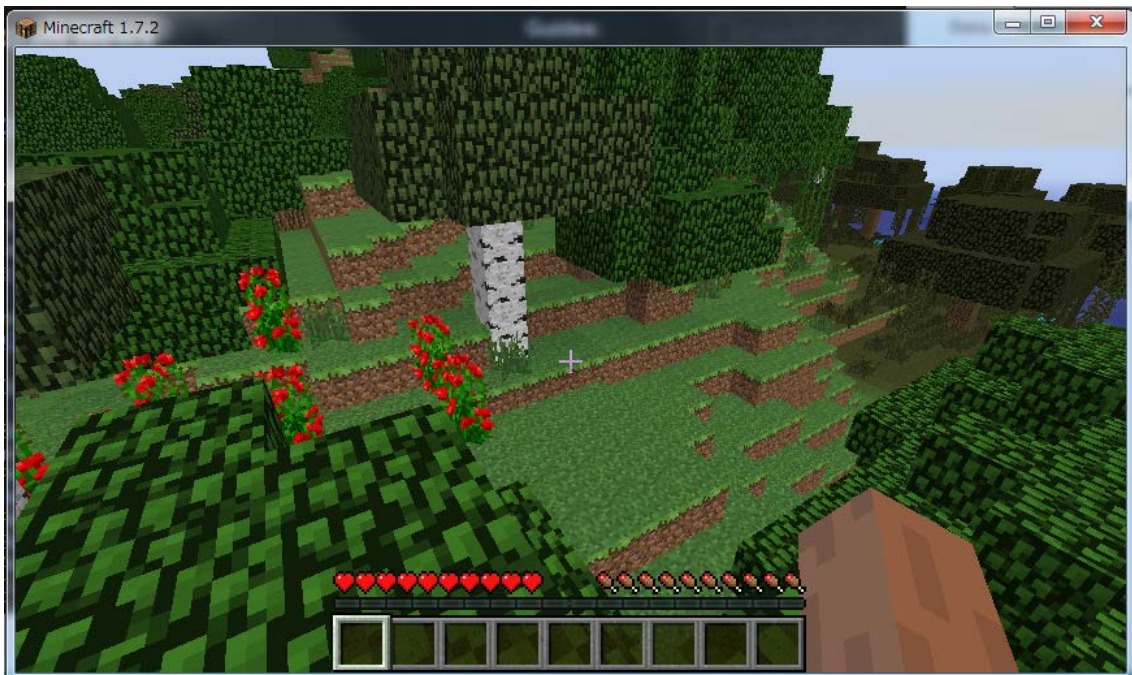


図 3 デフォルトゲーム画面



図 4 テクスチャ&スキン変更 MOD 使用例 像は考文[2]の引用元より

この様に MOD を導入することでゲームユーザーは既存のゲームを全く別のゲームへと変化させることができる。これによりユーザーの選択の幅が広がり、ゲームに対する興味

を維持することができると予想できる。

しかし、前提 MOD にも種類があり、MOD 自体個人で制作している物であるため、どんな MOD でも対応しているというわけではなく、それぞれの前提 MOD に互換性が無い、相性が悪い等でゲーム自体が起動しない、クラッシュしてしまう可能性が存在するためしっかりとそれぞれの MOD の相性を確認しておくことが必要である。

2.5 MOD の導入方法

ここでは Minecraft における MOD の導入方法について記述していこうと思う。

① Minecraft のバージョンを確認する

これは MOD がその Minecraft のバージョンに対応しているか確認するために必要な作業である。

② Minecraft Forge を導入する

Minecraft Forge とは上記にも述べた通り、MOD を導入するにあたっての前提 MOD である。これは、Minecraft Forge の公式サイトが存在するため、そこからダウンロードする。(図 5 参照)

Version	Minecraft	Time	Downloads
10.13.1.1226	1.7.10	10/14/2014 08:04:12 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.1.1225	1.7.10	10/11/2014 02:38:52 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.1.1224	1.7.10	10/08/2014 09:02:14 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.1.1223	1.7.10	10/08/2014 08:58:16 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.1.1222	1.7.10	10/06/2014 06:34:54 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.1.1221	1.7.10	10/06/2014 06:15:54 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.1.1220	1.7.10	10/06/2014 05:54:28 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.1.1219	1.7.10	10/02/2014 08:04:57 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.1.1217	1.7.10	09/14/2014 06:25:24 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1208	1.7.10	08/20/2014 04:05:27 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1207	1.7.10	08/15/2014 05:00:01 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1206	1.7.10	08/14/2014 11:37:11 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1205	1.7.10	08/13/2014 12:03:57 AM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1204	1.7.10	08/12/2014 08:36:25 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1203	1.7.10	08/12/2014 02:47:23 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1202	1.7.10	08/11/2014 10:22:57 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1201	1.7.10	08/11/2014 07:04:08 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1200	1.7.10	08/10/2014 04:45:57 AM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1199	1.7.10	08/05/2014 10:52:08 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1198	1.7.10	08/04/2014 05:39:03 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1197	1.7.10	08/03/2014 10:11:38 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1195	1.7.10	08/03/2014 06:51:34 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1194	1.7.10	08/03/2014 06:51:43 AM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1191	1.7.10	08/01/2014 09:41:11 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *
10.13.0.1190	1.7.10	07/30/2014 04:15:26 PM	(Changelog) (Installer) * (Installer-Win) * (Javadoc) * (Src) * (Universal) * (Userdev) *

図 5 Minecraft Forge 画面

図 5 の場合、Minecraft のバージョンを確認した後、自分の欲しいバージョンの Minecraft Forge のページまで行き (今回はバージョン 1.7.10) Installer をクリックすると Minecraft Forge のダウンロードが開始される。ダウンロードされたもの (Minecraft Forge Installer) を起動すると図 6 の画面になる。



図 6 インストーラー画面

OK を押しその後コンプリートと出れば、Minecraft Forge の導入は完了。

- ③ PC の検索で「%appdate%」を検索する

「appdate」フォルダから「.minecraft」フォルダへ移動し、「Mods」というフォルダが新しく作られている。(図 7 参照)

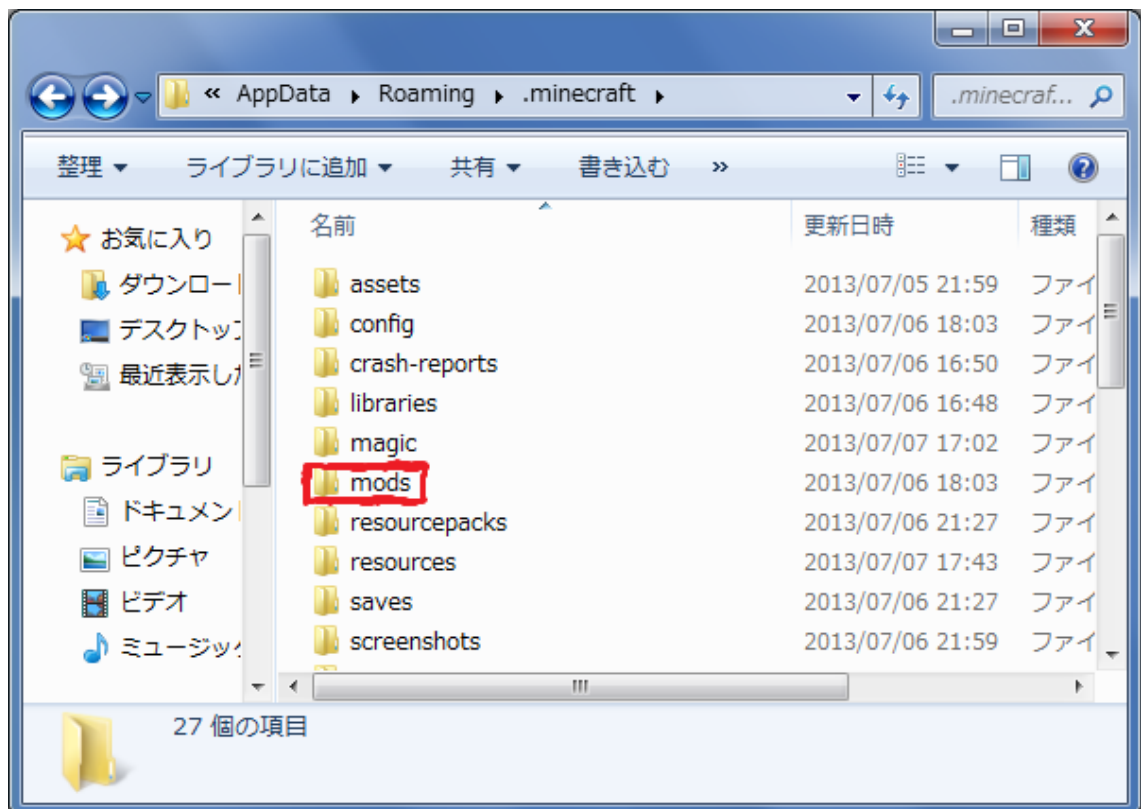


図 7 mods フォルダ追加後

④ ダウンロードした MOD データを mods フォルダへ入れる

好みの MOD データを mods フォルダへ入れた後 Minecraft を起動すれば MOD 導入の完了である。この場合に注意しなければならないことは上記にも述べた通り，MOD に相性があるため事前に調べておく必要があるということである。

参考文献

- [1] *Strawberry-candle*. 【マイクラ珍道中 No.5】よせばいいのに影 Mod 導入. 2012-07-31.
<http://st6candle.blog121.fc2.com/blog-entry-500.html> (参照 2014-09-17).
- [3] FPS†ZH. L4D2 マインクラフト スキン MOD. 2012-09-29.
<http://321cbnfg.blog20.fc2.com/blog-entry-283.html> (参照 2014-09-17).

第三章

GitHub について

3.1 本章の構成

本章では本研究で Minecraft MOD を調査するにあたり，利用するバージョン管理システムである GitHub についての基本知識，GitHub の API の解説等を記述していく．

3.2 バージョン管理システムについて

バージョン管理システムは，ファイルの履歴を管理するシステムであり，修正や追加など作業によって生成されたファイルについての複数の履歴を記録し，後から古い履歴の取り出しや，差分の参照が可能になるシステムである．これらのファイルの更新履歴をリポジトリと呼び，自分が作業したファイルの更新をリポジトリに反映させることをコミットすると言う．またバージョン管理システムソフトウェアによっては，ファイルの DELETE や移動の履歴を確認する機能や，特定の利用者がファイルの管理する権限を獲得するロック機能，複数の変更を統合するマージ機能がある．

開発プロジェクトにおいて複数人で同一のファイルを編集する必要があるとき，バージョン管理システムの機能が役立つのである．バージョン管理システムを利用すると，更新者や変更点，変更日時が確認できるため，誰がいつどこを編集を行ったのかすぐに理解することが出来，混乱や時間の無駄遣いを避けることが出来，とても効率的に作業が進められるのである．しかし，問題点として複数の人間が複数のファイルを各々編集するため，それぞれのファイルの最新の状態が分からなくなり，同一ファイルに対する変更が競合するなどの問題が生じやすい点がある．

ソフトウェア開発においては，バージョン管理システムは特に有効的である．ソースコードなど長い文を編集した際など変更点を容易に把握でき管理できるので，障害がいつ発生したのか，どの時点から問題となっていたのか，いつ修正されたのかななどを容易に調べることが出来，早期の問題解決が可能になる．

また，バージョン管理システムは，管理方法の違いにより 3 つに分類される．ファイル単位で個別バージョン管理を行う「個別バージョン管理システム」，リポジトリをサーバーで一元管理し，コミットなどの操作はメンバーが行う「集中型バージョン管理システム」，リポジトリをメンバーで管理でき，そのメンバー間でリポジトリの連携が可能である「分散型バージョン管理システム」である．

3.2.1 集中型バージョン管理システムについて

集中型バージョン管理システムは Git が登場する前から使われているバージョン管理システムである．集中型バージョン管理システムよりも先に公開されたバージョン管理シス

テムに個別バージョン管理システムがあるが，集中型バージョン管理システムは個別バージョン管理システムを参考にして開発された．そのため，まず集中型バージョン管理システムについて述べていこうと思う．

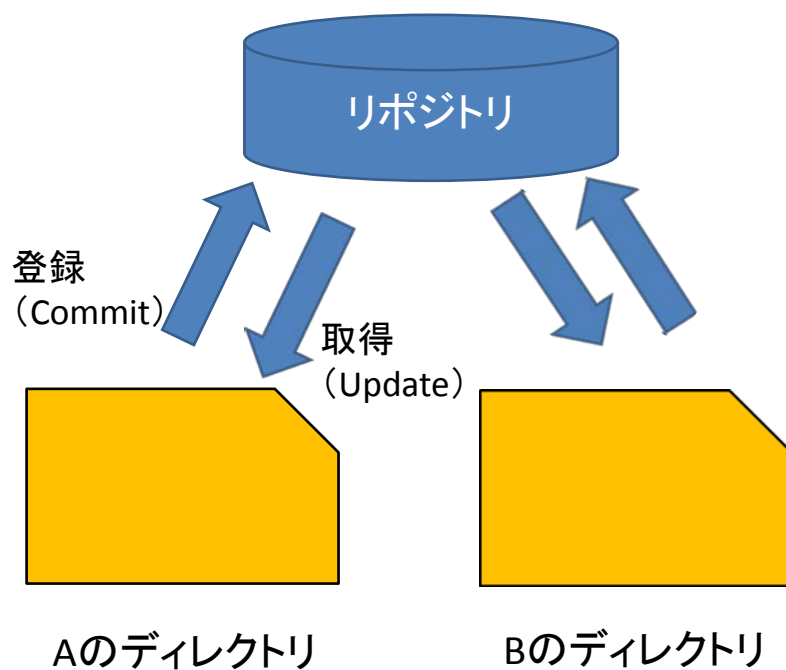


図 8 集中型バージョン管理システム

集中型バージョン管理システムでは，1つのソフトウェアに対して1つのリポジトリを使用する．ソフトウェア開発に参加するメンバーは，中央リポジトリ（プロジェクトメンバー間で共有するリポジトリ）からソースコードを取得し編集する．編集が終了した時，ソースコードを中央リポジトリに登録する．この集中型バージョン管理システムはリポジトリのあるサーバーに接続できない環境である場合，最新のソースコードを取得することや，ファイルの編集を反映させることはできない．

3.2.2 分散型バージョン管理システム

3つの種類のバージョン管理システムが存在すると前述したが，ここでは本論文の調査環境である GitHub に用いられるバージョン管理システムである，分散型バージョン管理システムについて述べる．

分散型バージョン管理システムではリモートリポジトリをサーバー上に設置し，プロジェクトのメンバーがそれぞれの端末にローカルリポジトリを持つという構成が考えられる．

そのため、コミットの参照や差分を所得する場合に手元にあるリポジトリにアクセスしなければならないが、この場合ネットワークに接続している必要がないのである。またローカルファイルにリポジトリが存在するため、高速に動作することが可能であり、変更点を送受信する仕組みがあるため、複数のリポジトリ間で連携することが出来る。ソフトウェア開発において開発者同士の作業を柔軟に進めることができる、この分散型バージョン管理システムはオープンソース開発に適している。そして、その分散型バージョン管理を行えるのが **Git** である。分散型バージョン管理システムを図は以下である。

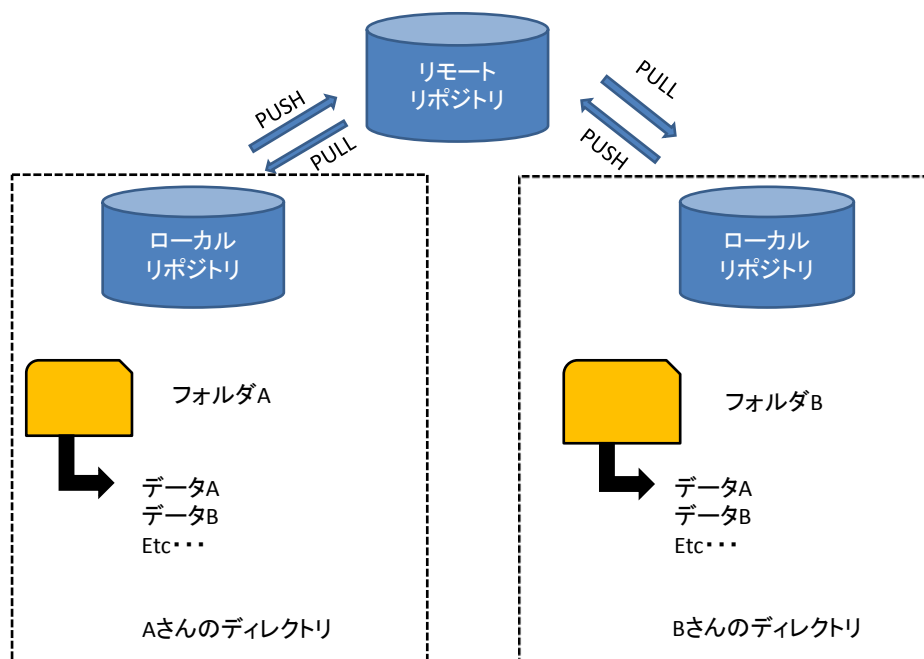


図 9 分散型バージョン管理システム

3.3 Git について

分散型バージョン管理システムである **Git** は 2005 年、Linux カーネル開発現場での必要性から開発が始まった。それまで、Linux カーネル開発のソース管理には **BitKeeper** というバージョン管理システムが用いられていたのである。これは **BitMover** 社製の商用のバージョン管理システムである。**Bitkeeper** は、先進の分散型バージョン管理システムで、カーネルプロジェクトが採用した当時、オープンソースの世界にはこれに匹敵する分散型バージョン管理システムの使用が不可欠であった。そのため、Linux は **Git** を開発したのである。

Git は分散型バージョン管理システムであるため、サーバーを必要としない。またユーザーそれぞれのコンピュータ上にリポジトリを持ち、それぞれが互いに連携しあうことがで

きる。さらに、基本的なそれぞれのリポジトリにすべての履歴が保存されるため、差分やログの表示などを高速に行えるのである。

リポジトリ間連携はネットワーク通信やメールを経由して行う。ほかにもリポジトリを共有リポジトリとして公開する仕組みや、ユーザー管理と組み合わせる方法があり、集中型バージョン管理システムのような利用形態もとれる。

また、他のバージョン管理システムとのデータ交換も可能であり、既存の他のバージョン管理システムのリポジトリを **Git** リポジトリへ変更することや、中央リポジトリに他のバージョン管理システムのリポジトリを利用し、手元では **Git** を利用する、といった形態をとることもできるのである。

オープンソースのバージョン管理システムとしては、**CVS** や **Subversion** が有名で、今でもこれらの集中型システムはよく使われている。しかし、近年になって、**Linux** カーネル、**X.org**, **Ruby on Rails**, **Perl** といった有名なプロジェクトが **Git** に乗りかえて成功裡に使用しているのをみて、**Git** を使用し始めとする分散型バージョン管理システムを使用するプロジェクトは飛躍的に増加してきている。

3.3.1 **Git** の特徴

Git は主にファイル自身、ファイルの集合としてのツリー、そしてコミット情報という 3 つの情報を管理している。それぞれの情報はハッシュ値をもとに管理され、このハッシュ値はファイルが同一かどうかの判断にも用いられる。**Git** はコミット情報を差分管理ではなく、ファイルそのままを保持しているという特徴を持っている。さらに時間的な変遷を管理する仕組みや、コミットをメールで受信する仕組みなどがある。また、ローカルコンピュータ上にリポジトリを持つため、場所や時間、あるいはネットワーク接続の状態に関わらずコミットすることが出来る。

3.4 **GitHub** とは

GitHub とは、**GitHub.com** により運営されている **Git** ホスティングサイトである。**Git** リポジトリを利用してプロジェクトやソースコードの管理を行うバージョン管理システムを提供しているサービスである。**GitHub** は **Git** ホスティングサイトとしては最も多く利用されているサービスであり、170 万を超える人が利用している。

また **GitHub** はソーシャルな機能が特徴で、プログラマー同士がコードの共有を行ったり、コードの公開をし合ったりしている。そして、**GitHub** ではソースコードだけでなく、画像やドキュメントなど、どんなファイルでもアップロードすることが出来、管理することが出来る。

3.4.1 GitHub の基本用語

GitHub の基本用語について以下に記述する。

表 1 GitHub の用語説明

用語名	内容
リポジトリ	システム開発におけるソースコードの貯蔵庫のようなもの。一種のデータベースであり、ソフトウェア開発および保守における各工程の様々な情報を一元管理する。
コミット	ファイルの変更履歴情報を閲覧したり、ファイルの変更を保存したりすることである。
共有リポジトリ	共有リポジトリとは、チームメンバーで共有するリポジトリで、ソースコードのメインバージョンが保存されている。
ローカルリポジトリ	作業者のコンピュータ上にあるリポジトリである。
インデックス	ローカルリポジトリへ反映する変更を一時的に保存しておく場所である。インデックスの内容は、 <code>commit</code> によりローカルリポジトリへ反映される。
作業ツリー	ローカルリポジトリ上にある現在の作業ファイルである。作業ツリーの変更点は <code>add</code> によるインデックスに追加される。
ディレクトリ	フォルダのことである。ファイルを分類・整理するための保管場所である。
フォロー	特定のユーザーをフォローすることが出来る。
フォロワー	ユーザーをフォローしているユーザー。
スター	SNS で使われている「いいね!」や「good」のようなもの。つけられたスターはカウントされ、スターのカウントが多いとほかのユーザーからの注目をされていると認識できる。
リビジョン	ある期間内までの過去のプロジェクトデータやある程度まとまったプロジェクトデータを記録したものである。
ウィキ (Wiki)	その場にページが表示され、ドキュメントやコードがかかる場所である。
Oragnization	Oragnization とは普段、個人的に GitHub を使用している人が仕事などで GitHub を仕事用に使用したい場合にアカウントをもう 1 つ作成するのではなく、同じアカウントで会社用に使用するアカウントとして使用するのが Oragnization である。

フォーク (forking)	1 つのプロジェクトが複数に分岐していくことである。だれかのリポジトリをほかの人がコピーし改変していくことである。
ライト (write)	フォークしたときにコピーしたオリジナルのリポジトリのデータに書き込みをすることである。
プルリクエスト	フォークを行い、コピーしたリポジトリのデータにライトしたことをオリジナルのリポジトリのユーザーに通知を POST することである。
マージ (merge)	プルリクエストした人がその人のリポジトリに対して行われた変更を自分のリポジトリにも取り入れることである。
イシュー (Issue)	Issue とは、1 つのタスクを 1 つの Issue に割り当てて、データの監視や管理を行えるようにするための機能である。1 つの機能変更や修正などに対して 1 つの Issue が割り当てられるため、Issue を見れば、そのタスクの変更や修正に関することがすべてわかるよう管理できるのである。また、イシューにタグやマイルストーンをつけることも可能である。タグ機能は初期の設定の場合では、「バグ」、「重複」、「強化」、「無効」、「質問」が設定されている。タグの種類は増やすことも可能である。また、Issue には、「Open」と「Close」機能があり、Issue を受信した人は受信した Issue を拝見したら Open をクリックし、拝見し終わったら、Close をクリックすることで、Issue を発行したユーザーに拝見し終わったことを通知することができる。
ウォッチ機能	他の人のデータを見ることができる。他の人の進捗状況やプロジェクト内容を閲覧できる。
グループ機能	特定のアカウントユーザーで構成し、特定のユーザー同士でリポジトリを共有したりすることができる。グループ機能を公開の状態にしておけば、グループ以外のユーザーも閲覧はできるが、グループ機能を非公開の状態にするとグループで決められたユーザー以外は閲覧やリポジトリを操作することができないように設定できる。
検索機能	検索機能は「検索ウィンドウ」に検索ワード（ユーザー名やプロジェクト名、コードなど）を入力するとそれに関連した情報を表示することができる。
ブランチ	バージョン管理システムの管理下にあるオブジェクトを複製し、それぞれ同時並行して変更が行えるようにしたものこと。
HEAD	今いるブランチの最新のコミットへのポインタ。

リネーム	ファイルやフォルダ(ディレクトリ)などの名前を変えること.
------	-------------------------------

3.4.2 Git コマンド

GitHub は Git ホスティングサービスである. そのため, リポジトリの作成・運用を, Git コマンドを用いて行うことができる. リポジトリを操作するうえで必要な Git コマンドは以下である.

- `git push`

リモートリポジトリに自分の内容を送信する.

```
git push <repository> <refspec>
```

<refspec>を省略すると、デフォルトではリモートリポジトリとローカルリポジトリの双方に存在するブランチが対象となる.

```
git push <リモート送信先リポジトリ> <ローカル送信元ブランチ>:<リモート送信先ブランチ>
```

使用例) `topic-branch` ブランチの内容を `origin` サーバーにプッシュしたい場合

```
$ git push origin topic-branch
```

付随するすべてのコミットおよび内部オブジェクトと共に指定したブランチを <remote> にプッシュするコマンド. このコマンドを実行すると, プッシュ先のリポジトリにローカルブランチが作成される. 変更の誤書き込みを防止するために, Git ではプッシュ先リポジトリにおける統合処理が早送りマージ以外である場合にはプッシュが拒否されます.

```
git push <remote> --force
```

上のコマンドと同様であるが, 早送りマージ以外の場合にも強制的にプッシュが実行される. プッシュ操作によって何が起こるかを完全に理解している場合以外は `--force` フラグを使用してはいけない.

```
git push <remote> --all
```

すべてのローカルブランチを指定したリモートリポジトリにプッシュするコマンド。

```
git push <remote> --tags
```

ブランチをプッシュしただけでは、例え--all フラグが指定されていても、タグは自動的にプッシュされない。そのため--tags フラグを指定することにより、すべてのローカルタグをリモートリポジトリに送ることができる。[5]

- git branch

```
git branch [topic-branch]
```

現在のブランチを元に新しいブランチが作られる。

現在のソースツリーを元に新たなブランチを作成するには、「git branch」を使用する。また、操作対象とするブランチを切り替えるには、「git checkout」を使用する。

たとえば、新たに「temp01」というブランチを作成し、以後このブランチで作業を行うには次のようにする。

```
$ git branch temp01
$ git branch
* master
  temp01
$ git checkout temp01
Switched to branch "temp01"
$ git branch
  master
* temp01
```

なお、git checkout に「-b」オプションを付けて実行すると、新たにブランチを作成してそのブランチに切り替える、という作業が一発で行える。たとえば、「git checkout -b

temp01」は、次の2つのコマンドを順に実行した場合と同じ結果になる。

```
$ git branch temp01
$ git checkout temp01
```

どのブランチがどのブランチを元に作られたか、という情報は「git show-branch」で確認できる。git show-branch では次の例のような出力を行うが、前半の「---」までがブランチ履歴、「---」から後半が最近のコミット履歴を表している。

```
$ git show-branch
! [master] add script merging google anal.'s log and OTP's log.
! [temp01] Cleaning codes and convert TAB to space.
* [temp02] Cleaning codes and convert TAB to space.
---
+* [temp01] Cleaning codes and convert TAB to space.
+* [temp01^] Cleaning code, and add comments
++* [master] add script merging google anal.'s log and OTP's log.
```

前半部分ではブランチが階層構造で表示されており、「*」が付いている行が現在の作業ブランチになる。また、[] 内の文字列はそれぞれのコミット単位を表すショートネームである。この例では、「master」から「temp01」ブランチが作成され、さらに「temp01」から「temp02」ブランチが作成されたことが分かる。

```
git branch [topic-branch] origin/develop
```

リモートの origin/develop を起点に新しいブランチが作られる。この場合、自動で origin/develop 追跡ブランチの設定も行われる。

- git checkout

操作対象とするブランチの切り替え。リポジトリに存在するブランチをワーキングツリーに展開する。その場合ステージエリアも切り替わる。-b オプションを付けた場合、新たにブランチを作成し、そのブランチに切り替える。ということになる。

使用例)ローカルブランチ名を指定してリモートブランチをチェックアウトする

```
$ git checkout -b [other_branch] [origin/other_branch]
```

- git fetch

ローカルのリモート追跡ブランチをリモートのブランチと同期させること。リモートリポジトリにあるブランチの最新情報をローカルリポジトリに取得すること。Pullのようにファイルが更新されるわけではなく、ローカルリポジトリが更新されるだけである。

例)ローカルブランチ名を指定してリモートブランチをチェックアウトする

```
$ git checkout -b [other_branch] [origin/other_branch]
```

- git merge

現在の作業ブランチに別のブランチで行われた変更点を取り込む。

使用例) リモートブランチと同期したデータ、追跡ブランチをローカルリポジトリに取り込む

```
$ git merge origin/other_branch
```

マージが成功した場合はそのままコミットも行われる。いっぽう、競合が発生したファイルには下記のような形式で競合している個所にマーカが埋め込まれる。

```
<<<<<<< <ブランチ名>:<ファイル名>
```

```
<作業中ブランチの内容>
```

```
=====
```

<変更点の取り込み元（マージ元）ブランチの内容>

>>>>>> <ブランチ名>:<ファイル名>

たとえば、下記は「markup.pl」というファイルに対してマージを行い、競合が発生した例である。「<<<<<<< HEAD:markup.pl」から「=====」までが作業ブランチ内の markup.pl に記述されていたもので、「=====」から「>>>>>>> e74597cbfdb9995e540ca9e8c8a6e79705e2889c:markup.pl」までがマージ元ブランチ内の markup.pl に記述されていたものとなる。

```
sub ulist {
    if( $l =~ m/^\☆(リスト.*)$/ ) {
        $cap = $1;
    }

<<<<<<<< HEAD:markup.pl
    print "<ul>¥n";
    while( $l =~ m/^\cdot / ) {
        $l =~ s/^\cdot(.*)$/<li>$1<¥/li>/;
        print "foo:$l¥n";
        $l = <>;

=====
    print "<p><b>$cap</b></p>¥n";
    print list_start( $cap );
    while( $l = <> ) {
>>>>>>> e74597cbfdb9995e540ca9e8c8a6e79705e2889c:markup.pl
        chomp $l;
        $l =~ s/&/&amp;/g;
        $l =~ s/</&lt;/g;
        $l =~ s/>/&gt;/g;

    }
```

なお競合が発生した場合、このマーカーを取り除いて競合を解消するか、もしくはマージを取り消すまでコミットが行えないので注意しなければならない。

- git rebase

ブランチの派生元（上流）を変更する.

`git rebase <派生元ブランチ>`

カレントブランチの履歴が派生元ブランチの後ろに付け替えられ、履歴が一本化される

コンフリクトした場合、解消後に以下コマンド

`$ git rebase --continue`

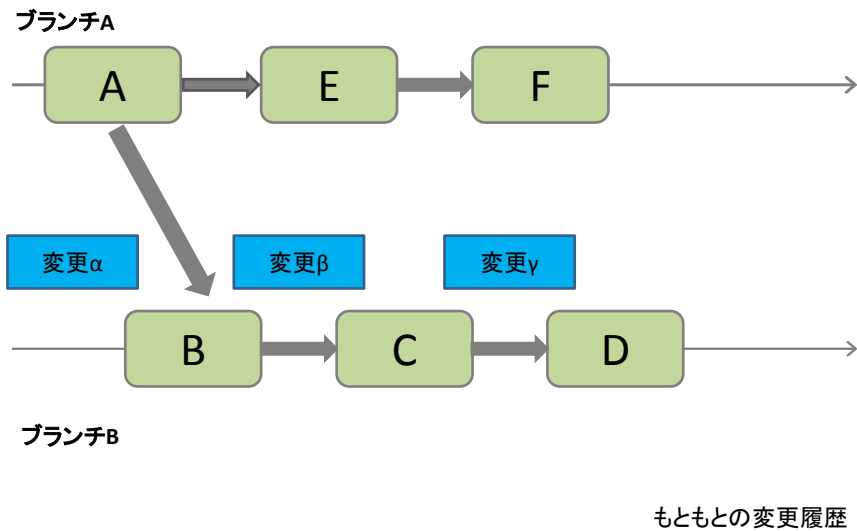


図 10 ブランチの git rebase 処理前

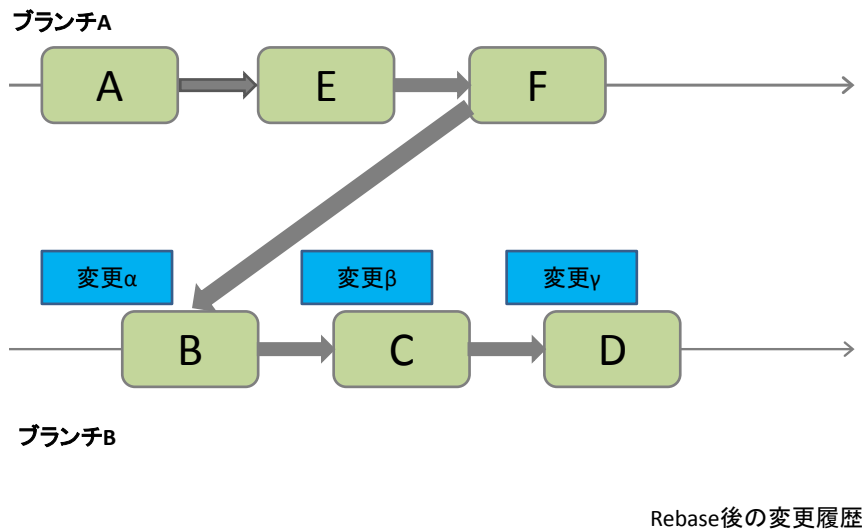


図 11 ブランチの git rebase 処理後

図 3-3 の例の場合、ブランチ A から派生したブランチ B に対して、「変更 α」と「変更 β」、「変更 γ」という 3 回のコミットを行っている。また、ブランチ B を作成したのち、ブランチ A には複数のコミットが行われている。このような状態でブランチ B に対し「get rebase ブランチ A」を実行すると、図 3-4 の様にブランチ A の最新版に対して、ブランチ B を作成してからブランチ B の最新版までの間に加えられた変更（この場合変更 α と変更 β、変更 γ）が適用され、それがブランチ B の最新版となる。

- git pull

git fetch でローカルの追跡ブランチにリモートブランチのデータを反映し、トピックブランチに merge する。ほかのリポジトリで加えられた変更点を現在のブランチにマージするには、「git pull」コマンドを利用する。

```
$ git pull <変更点の取り込み元リポジトリ>
```

変更点の取り込み元リポジトリは URL で指定する。

```
$ git pull
```

現在のブランチの指定したリモートにおけるコピーをフェッチして、それを現在のブランチに即時マージする。これは、`git fetch <remote>` コマンドを実行し、続いて `git merge origin/<current-branch>` コマンドを実行するのと同等である。

```
git pull <remote>
```

上のコマンドと同様ですが、リモートブランチを現在のブランチにマージする際に `git rebase` コマンドを使用する。

- **git init**

リポジトリを作成する。Git を利用するには、まずリポジトリの作成が必要である。リポジトリ作成を行うコマンドは「`git init`」である。リポジトリを作成したいディレクトリでこのコマンドを実行すると、`.git` ディレクトリが作成され、Git リポジトリの管理ファイル等がここに作成される。

```
$ cd <リポジトリを作成するディレクトリ>
$ git init
```

- **git clone**

既存のリポジトリの複製を作る。リポジトリの複製を行うには「`Git clone`」コマンドを利用する。このコマンドは既存のリポジトリからローカルにファイルをコピーして作業用のリポジトリを作成する、といった場合などに利用する。

```
$ cd <リポジトリを作成するディレクトリ>
$ git clone <複製したいリポジトリの URL>
```

リポジトリの URL は表 2 のような形式で指定する。

表 2 Git で利用できるリポジトリ URL

プロトコル	指定方法
rsync	<code>rsync://<ホスト名>/<Git リポジトリのパス></code>

HTTP	http://<ホスト名>[:ポート番号]/<Git リポジトリのパス>
HTTPS	https://<ホスト名>[:ポート番号]/<Git リポジトリのパス>
git	git://<ホスト名>[:ポート番号]/<Git リポジトリのパス>
SSH	ssh://[ユーザー名@]<ホスト名>[:ポート番号]/<Git リポジトリのパス>
ローカルファイル	<Git リポジトリのパス>もしくはfile://<Git リポジトリのパス>

また、作成するディレクトリ名を指定することもできる。

```
$ git clone <複製したいリポジトリの URL> <作成するディレクトリ>
```

git clone はリポジトリの履歴情報までも含めてリポジトリを複製するため、大規模なプロジェクトの場合は多くのファイルを取得/コピーしなければならないことがある。もし最新のソースファイルを取得する目的のみで使用し、ソースファイルの修正を行わない場合は、「--depth」オプションで取得するリビジョン数を指定するとよい。たとえば最新リビジョンだけを取得したい場合、「--depth 1」と指定する。

- git fsck

リポジトリの正当性チェックを行う。リポジトリがもし破損した可能性がある場合、「git fsck」で破損している個所を検出できる。

```
$ git fsck
```

また、完全なチェックを行うには「--full」オプションを付けて git fsck を実行する。

```
$ git fsck --full
```

「git fsck」では、「dangling オブジェクト」と呼ばれるテンポラリオブジェクトも検出される。dangling オブジェクトはリポジトリに変更を加える際に作成される中間ファイルのようなもので、たとえば「git add」で追加したファイルに別の変更を加えてコミットした場合などに残されるものだ。このファイルはストレージ容量を消費するもの

の、残っていても大きな害はないため、無視して構わない。また、後述の「`git gc`」コマンドを実行することで、一定期間（デフォルトでは 2 週間）よりも前に作成された `dangling` オブジェクトを破棄することができる。

- `git gc`

リポジトリ内の不要なオブジェクトを削除し、最適化を行う。「`git gc`」コマンドは、リポジトリ内のオブジェクトの圧縮 (`pack`) や不要なオブジェクトの破棄などを行うものである。これによってリポジトリが使用するストレージ容量を減らせるだけでなく、パフォーマンスの向上も期待できる。頻繁に実行する必要はないが大量のコミットやマージを行った場合などには実行するとよい。

```
$ git gc
```

- `git status`

変更が加えられたファイルを表示する。

- `git diff`

ファイルに加えられた変更点を `diff` 形式で表示する。

- `git add`

コミットするファイルを指定する。

- `git commit`

変更点をコミットする。

リポジトリに新たにファイルを追加し、変更を加えたファイルを指定するのに利用するコマンドが「`git add`」である。どのファイルが追加/変更されたのかは、「`git status`」で確認できる。`git status` を実行すると、次の例のように追加/変更されたファイルの情報が表示される。

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   HBPreferencesTransformer.h
#       modified:   HBPreferencesTransformer.m
#
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

特定のファイルにどのような変更が加えられたかは「git diff」コマンドで確認できる。

```
$ git diff <変更を確認したいファイル>
```

変更点を保存するには、git add コマンドで対象とするファイルを指定したのちに「git commit」コマンドを実行する。

```
$ git add <追加/変更したファイル 1> <追加/変更したファイル 2> ...  
$ git commit
```

また、git commit を「-a」オプションを付けて実行すると変更が加えられたファイルを自動検出してコミットできる。ただし、この場合新規に作成されたファイルはコミット対象にはならない。

```
$ git commit -a
```

通常、git commit コマンドを実行するとエディタが起動してコメントメッセージの入力が求められる。エディタを起動したくない場合は「-m」オプションでメッセージを指定すればよい。

```
$ git commit -m "<コミットメッセージ>"
```

- git log

コミットログを閲覧する 「git log」でリポジトリのコミットログを閲覧できる。コミットログは新しいものから順に表示され、次の実行例のようにそれぞれのコミットについてそのコミットを表すハッシュ値とコミット者、タイムスタンプ、コメントが表示される。

```
$ git log  
commit 0b70750bdf9b02597741301c695ff46bc75036d4  
Author: hoge <hoge@users.sourceforge.jp>  
Date: Thu Mar 12 19:47:31 2009 +0900
```

```
Cleaning codes and convert TAB to space.
```



```
commit 769735fbce0a0a23a2b7547003d43518b4ec96f3
```

```
Author: hoge <hoge@users.sourceforge.jp>
```

```
Date: Thu Mar 12 19:43:09 2009 +0900
```

```
    Cleaning code, and add comments
```

```
    :
```

```
    :
```

- **git reset**

直前のコミットを取り消す。コミット後に小さなミスなどに気付いた場合などに便利なのが「git reset」である。たとえばコミット後に「git reset --soft HEAD^」と実行すると、直前に行ったコミットを取り消すことができる。

```
$ git reset --soft HEAD^
```

この場合、作業ツリーの内容はコミット時のままで「コミットした」ということだけが取り消される。もし作業ツリーに加えた変更点までも取り消したい場合は、「--soft」の代わりに「--hard」を指定すればよい。

また、コミットされていた内容は「**ORIG_HEAD**」という名前で参照できる。これを利用して、次のように実行することで現在の状態とコミットされていた状態の差分を表示できる。

```
$ git diff ORIG_HEAD
```

修正後に再度コミットを行うには、下記のようにする。これを実行すると、先に入力したコミットメッセージを再編集してコミットが行える。

```
$ git commit -a -c ORIG_HEAD
```

なお、git commit に「--amend」オプションを付けて実行することで、直前に行っていたコミットを訂正することができる。

```
$ git commit
```

(コミット後にミスに気づき、ミスを修正)

```
$ git commit --amend
```

 (先に行ったコミットが、ミスを修正したものに置き換えられる)

上記の操作は、次のような操作とほぼ同一の働きをする。

```
$ git commit
```

```
$ git reset --soft HEAD^
```

 (コミット後にミスに気づき、コミットを取り消し)
(ミスを修正)

```
$ git commit -c ORIG_HEAD
```

- **git revert**

作業ツリーを指定したコミット時点の状態にまで戻す。「**git revert**」は作業ツリーを指定したコミット時点の状態にまで戻し、コミットを行うコマンドである。引数にはコミットを指定するハッシュ文字列もしくはタグ名などを指定する。

```
$ git revert <コミット名>
```

git revert は **git reset** と似ているが、作業ツリーを差し戻したという情報が作業履歴に残るのが異なる点である。

- **git tag**

Git ではコミットをハッシュ文字列で管理しているため、ユーザー側から見るとどの文字列がどのコミットを表しているのか分かりにくい。「**git tag**」は、直前のコミットに対して分かりやすい別名 (タグ) を付けるコマンドである。

```
$ git tag <タグ名>
```

タグ名を指定せずに **git tag** を実行すると、現在のリポジトリ内にあるタグを確認できる。

```
$ git tag rev1
```

 (「rev1」というタグを付ける)

```
$ git tag
```

 (タグを確認する)

```
rev1
```

- **git stash**

現在の作業ツリーの状態を一時的に保管する。現在の作業中の状態をコミットせずに、一時的にほかのブランチに対して作業を行いたい、という場合に役立つのが、「git stash」コマンドである。このコマンドでは、現在の作業ツリーの状態を一時的に保存しておくことができる。

```
$ git stash <保存名もしくはコメントなど>
```

git stash の実行後は、git checkout などで作業したいブランチをチェックアウトすればよい。作業の完了後作業中だった作業ツリーを再度呼び出すには、元のブランチをチェックアウトしてから「git stash pop」コマンドを実行する。また、「git stash list」を実行すると、一時保存されている作業ツリーが一覧表示される。

```
$ git stash list
$ git stash pop
```

- **git remote**

git remote は、他のリポジトリとの接続の作成、内容確認、削除を行うコマンドである。リモート接続とは、他のリポジトリへのダイレクトリンクではなく、ブックマークのようなものである。他のリポジトリにリアルタイムアクセスを行うのではなく、非短縮 URL への参照として使用可能な短縮名称として機能する。

```
git remote
```

他のリポジトリへのリモート接続の一覧を表示するコマンド。

```
git remote -v
```

上のコマンドと同様だが、各々の接続の URL が表示される。

```
git remote add <name> <url>
```

リモートリポジトリに対する新規接続を作成するコマンド。作成されると他の Git コマンドにおいて<url>の代わりに<name>を短縮ショートカットとして使用することができる。

```
git remote rm <name>
```

<name>という名称のリモートリポジトリへの接続を削除するコマンド.

```
git remote rename <old-name> <new-name>
```

リモート接続を<old-name>から<new-name>にリネームするコマンド.

- git reflog

Git では、reflog と呼ばれる機能が働いて、ブランチの先端に対する更新の追跡が行われている. これにより、いかなるブランチからいかなるタグからも参照されていない更新内容であってもその時点に戻ることができます. 履歴を書き換えた後であっても reflog にはブランチの過去の状態が記録されており、必要な場合にはそこに戻ることが可能である.

```
git reflog
```

ローカルリポジトリの reflog を表示するコマンド.

```
git reflog --relative-date
```

相対形式の日付(例: 2 週間前)で reflog を表示するコマンド.

参考文献

- [1] Incept Inc. “GitHub”. IT 用語辞典 e-Words. 2014-09-28. <http://e-words.jp/w/GitHub.html>, (参照 2014-09-29).
- [2] 江口和宏, 山本竜三, 株式会社ヌーラボ. “知らないで現場で困るバージョン管理システムの基礎知識 (1/3)”. @IT. 2013-05-20. <http://www.atmarkit.co.jp/ait/articles/1305/20/news015.html>, (参照 2014-09-29).
- [3] 平屋真吾, クラスメソッド株式会社. “ガチで 5 分で分かる分散型バージョン管理システム Git (3/6)”. @IT. 2013-07-05. http://www.atmarkit.co.jp/ait/articles/1307/05/news028_3.html, (参照 2014-09-29).
- [4] Incept Inc. “Git”. IT 用語辞典 e-Words. 2014-09-28. <http://e-words.jp/w/Git.html>, (参照 2014-09-29).
- [5] Atlassian. “Git チュートリアル”. Atlassian. <https://www.atlassian.com/ja/git/tutorial>, (参照 2014-10-09).

第四章

調査

4.1 本章の構成

本章では GitHub 内から MOD プロジェクトのデータを抽出するプログラムの解説,抽出したデータから作成したグラフを記述する.

4.2 調査環境・調査方法

本調査に必要である調査環境の構築を記述し, 本調査で目的とする MOD プロジェクトのデータを抽出するプログラムの解説, 抽出したデータからグラフを作成するために必要なプログラムの解説, それらを自動化したプログラムの解説記述を行う.

4.2.1 調査環境解説

本研究を行うために使用した OS は ubuntu である.

Ubuntu (ウブントゥ) とは, コミュニティ により開発されているオペレーティングシステムである. ラップトップ, デスクトップ, そしてサーバーに利用することができる. Ubuntu には, 家庭・学校・職場で必要とされるワープロやメールソフトから, サーバースフトウェアやプログラミングツールまで, あらゆるソフトウェアが含まれている.

Ubuntu は現在, そして将来に渡って無償で提供され, ライセンス料を支払う必要はない. Ubuntu をダウンロードすれば, その後完全無料で使用することができる.

Ubuntu は, セキュリティに配慮して設計されている. デスクトップおよびサーバーの無償セキュリティアップデートが, 少なくとも 9 ヶ月間に渡って提供される. 長期サポート (LTS)版を利用すれば, 5 年間に渡りセキュリティアップデート提供される. LTS 版を利用するために追加の費用は必要ない. すべての人が無償という同じ条件で, Ubuntu を使用することができる.

4.2.2 Ubuntu の特徴

目立った特徴としては Web を閲覧したり, メールを読み書きしたり, 文書や表計算をしたり, 画像を編集したり, その他さまざまなことができる. Ubuntu のデスクトップ CD には, 早くて簡単なグラフィカルインストーラが搭載されている. 一般的なコンピュータの

場合、25 分以内にインストールが完了する。

シンプルなデスクトップ	初期状態で、デスクトップにはアイコンがひとつもありません。デフォルトのデスクトップテーマは、目にやさしいものを使用しています。
すぐに使える	面倒な設定はすべて終えてあります。Ubuntu を一度インストールすれば、基本的なセットアップ作業は終わっているのですぐに使い始めることができます。
オフィスアプリケーション搭載	オープンオフィスには、他のオフィススイートと似たユーザインタフェースと機能が備わっています。また、よく使われる主要なデスクトップアプリケーションが含まれています。
様々な形式のファイルを編集可能	マイクロソフトオフィス、ワードパーフェクト、KOffice、StarOffice のファイルを開いて編集することができます。
手軽なアップデート	アップデートが利用できる時には、タスクバーのアップデートエリアに表示されます。単純なセキュリティフィックスから、完全なバージョンアップグレードまで、このエリアで通知されます。アップデート作業は簡単なので、マウスで何度かクリックするだけでシステムを最新の状態に保つことが可能です。
統合されたセキュアなプラットフォーム	ビジネスが成長するにつれ、ネットワークも大きくなります。より多くのアプリケーションが必要となり、多くのサーバーが要求されます。Ubuntu サーバーは、いくつかの共用設定をサポートし、一般的な Linux サーバの配置プロセスを単純化します。これにより、メール、Web、DNS、ファイル共有あるいはデータベースといった一般的なインターネットサービスをうまくまとめたプラットフォームを、早く簡単に新しいサーバーにセッ

	<p>トアップすることができます。</p> <p>Debian から受け継いだ鍵となる教訓は、デフォルトでセキュリティを保つことです。</p> <p>Ubuntu サーバーはインストール後どのポートも開いておらず、セキュアなサーバの構築に必要な不可欠なソフトウェアのみが導入されます。</p>
自動 LAMP インストールによる TCO 削減	<p>Ubuntu サーバーのインストールは約 15 分以内で終了し、LAMP(Linux, Apache, MySQL, PHP)サーバが起動して利用可能な状態にできます。Ubuntu サーバーだけのこの機能は、インストール時に利用できます。</p> <p>LAMP オプションにより、LAMP の各コンポーネントを別々にインストールして設定する必要がなくなります。各アプリケーションのインストールおよび設定方法を知っている方が数時間かかる作業を、自動的に行うことができるのです。LAMP オプションを使えば、十分なセキュリティ、インストール時間の削減、設定ミスリスクの軽減を実現することができ、その結果コストの削減につながります。</p>
ワークステーションをアップグレードするコストの削減	<p>Ubuntu サーバーには LTSP(Linux Terminal Server Project)を利用したシンクライアントサポートが含まれています。最新版である LTSP-5 により、単純なインストールと簡単なメンテナンスが提供されています。すべてのデータはサーバーに蓄積され、個々のワークステーションをアップデートし、セキュリティを保つよりも大幅にコストを削減することができます。Ubuntu のシンクライアントサポートによる利点には以下のようなものがあります。</p>
簡単な管理	<p>一つのシステムからすべてのクライアント</p>

	を管理できます. 新しいソフトウェアのインストール, それらの設定変更, サーバー上のあたらしいバージョンへのアップグレード, そしてすべてのクライアントをすぐに最新の状態にすることさえ可能です. 全てのクライアントのバックアップを1箇所で行うこともできます.
完全に自動化されたインストールとセットアップ	シンクライアントサーバーをインストールするのは, デスクトップシステムのインストールと同じくらい簡単です. 一度インストールを終えれば, サーバー上で管理作業を行うことなく新しいクライアントを加えることができます.
高速な障害復旧	クライアントシステムが故障しても, 他のシステムを使って仕事を継続することができます. 特別な設定は必要なく, すべてのユーザーデータと設定は保持されています.

(参考文献[1]から引用)

4.3 調査環境構築

- ① `curl` をインストールする.

```
sudo apt-get install curl
```

`Curl` を `ubuntu` にインストールする.

- ② `Github` のログイン情報を省略する.

```
echo 'ユーザ名:パスワード' > github.passwd
chmod 600 github.passwd
```

確認するために以下を行う.

```
cat github.passwd
```

`Github` のユーザ名とパスワードを表示させることが出来る.

- ③ `Issue` のデータを所得する.

```
curl -s -u ユーザ名:パスワード "https://api.github.com/repos/ユーザ名/リポジトリ名/issues"
```

④ ユーザ名, パスワードを省略したものを利用するためには以下のように実行する.

```
curl -s -u $(cat github.passwd) "https://api.github.com/repos/ユーザ名/リポジトリ名/issues"
```

⑤ API 呼び出し結果の内容を確認するには以下のようにパイプ | を利用する.

```
curl -s -u $(cat github.passwd) "https://api.github.com/repos/ユーザ名/リポジトリ名/issues" | grep title
```

⑥ Python と HTTP アクセスのための requests をインストールする。

```
sudo apt-get install python python-setuptools  
sudo easy_install requests
```

⑦ api.py を作業ディレクトリにダウンロードする.

api.py とは以下のようなプログラムで, データを所得し, ファイルに保存するプログラムである.

```
#!/usr/bin/python  
# coding: UTF-8  
  
import sys, json, requests  
  
tmp = open('github.passwd').readline().rstrip('\n').split(':');  
username = tmp[0]  
password = tmp[1]  
#print >> sys.stderr, username,password  
  
url = sys.argv[1]  
  
count = 0
```

```
while (url is not None):
    #print url
    r = requests.get(url, auth=(username, password))
    print >> sys.stderr, r.headers['status'],
    for item in r.json():
        count = count + 1
        print json.dumps(item)
    if (r.links.has_key('next')):
        url = r.links['next']['url']
    else:
        url = None
    print >> sys.stderr, count
```

⑧ JSON 形式のデータを扱う.

```
chmod +x jq
```

jq のバイナリを作業ディレクトリにコピーし, jq を実行できるようにする.

⑧ コマンドでグラフを描く.

```
sudo apt-get install gnuplot-x11
```

Gnuplot をインストールする.

Gunplot とは 2 次元または 3 次元のグラフを作成するためのコマンドラインアプリケーションソフトウェアのことであり, 入力した数式等をもとにグラフを作成することが出来る.

4.4 調査方法

GitHub から MOD プロジェクトの Issue データを取得する.

① Python と HTTP アクセスのための requests をインストールする.

```
sudo apt-get install python python-setuptools
sudo easy_install requests
```

② api.py をダウンロードする.

```
curl -s -u $(cat github.passwd)
https://raw.githubusercontent.com/taroyabuki/yabukilab/master/library/github/api.py > api.py
```

- ③ オープンな issues をすべて取得する.

```
python api.py
"https://api.github.com/repos/Diogenesthecynic/FullScreenMario/issues?per_page=100" >
openissues.txt
```

?per_page=100 は無くても動くが、一度に取得するデータを最大に設定しておくことで API の使用可能回数の消費を抑えたほうが良い.

>はリダイレクト. ここでファイル openissues.txt に結果を保存している. API でデータを取得するには時間がかかるため、一度ファイルを保存しておく方が良い.

- ④ クローズドな issues をすべて取得する.

```
python api.py
"https://api.github.com/repos/Diogenesthecynic/FullScreenMario/issues?per_page=100&state=closed" > closedissues.txt
```

- ⑤ コミット履歴をすべて取得する.

```
python api.py "https://api.github.com/repos/jquery/jquery/commits?per_page=100" > commits.txt
```

- ⑥ Jq の利用

```
sudo apt-get install jq
```

openissues.txt と closedissues.txt, commits.txt, events.txt を例に使い方を紹介する.

⑦ データの確認

less openissues.txt では結果が見にくい (less の使い方は自分で調べる. とりあえず q で終了.)

jq を使って, `jq 'openissues.txt | less` とすると, 整形して表示される. さらに `jq 'openissues.txt | less -R` とすれば, カラーになる.

⑧ コミット履歴

コミット ID とコミット日時を一覧表示するには `jq 'commit.committer.date,.sha' commits.txt`. この結果の偶数行目の時だけ改行するように整形すると, Excel などでの処理しやすくなる. そのためには, | (パイプ) で awk に渡して, 行数を 2 で割ったあまりがゼロ, つまり偶数行なら改行, 奇数行ならカンマを付ければよい. うまくできるようになったら, > commits.csv などをつけてファイルに保存する.

```
jq 'commit.committer.date,.sha' commits.txt | awk '{ printf("%s", $0); if (NR % 2 == 0) printf("\n"); else printf(","); }'
```

⑨ Issue

Issue のタイトルだけを一覧表示するには `jq 'title' openissues.txt` 結果を別ファイルに保存したければ>を使えばいい.

Issue の作成日時だけを一覧表示するには `jq 'created_at' openissues.txt`

タイトルは重複する可能性があるため, `jq 'id,.title,.created_at' openissues.txt` のように, ID をつけておく. この結果は, 行数を 3 で割ったあまりが 0 のときは改行を, それ以外 (つまり 1 か 2) のときはカンマを付けて整形する.

```
jq 'id,.created_at,.title' openissues.txt | awk '{ printf("%s", $0); if (NR % 3 == 0) printf("\n"); else printf(","); }'
```

> openissues.csv などをつけてファイルに保存し, Excel や Ubuntu の LibreOffice Calc で読み込む. その際, 2 列目を「日時」として読み込むようにする.

closedissues.txt から, オープン日時とクローズ日時, タイトルの一覧を作るには, 次のよう

にする。 > closedissues.csv などをつけてファイルに保存する。

```
jq '.id,.created_at,.closed_at,.title' closedissues.txt | awk '{printf("%s", $0); if (NR % 4 == 0) printf("\n"); else printf(","); }'
```

この結果を Calc で読み込んで、D1 に=(C1-B1)*86400 などと入力して D 列全体にコピーすれば、issue がオープンしてからクローズするまでの時間を求められる。

⑩ ラベルの取得

ラベルは配列になっているため、取得する時には `jq '.id,.created_at,.labels[].name' closedissues.txt` のように、`[]` という記法を用いる。ただ、付いているラベルの数によって形式が変わってしまうため、これでは結果を次の処理に回しにくい。 `jq '.id,.created_at,[.labels[].name]' closedissues.txt` なら形式は揃うが、使いにくいことには変わらない（`-c` は 1 行 1 オブジェクトにするオプション）。

```
jq -c '{id,created_at,label:.labels[].name}' closedissues.txt
```

とすれば、ラベルの分だけオブジェクトができる（ラベルがないと出てこない）。

```
jq -c '{id,created_at,label:.labels[].name}' closedissues.txt | jq '.id,.created_at,label' | awk '{ printf("%s", $0); if (NR % 3 == 0) printf("\n"); else printf(","); }'
```

⑪ 個人の活動

個人の活動の日時とタイプを一覧表示するには `jq '.created_at,.type' events.txt`。先の例と同様に、結果を整形すると使いやすい。うまくできるようになったら、> events.csv などをつけてファイルに保存する。

```
jq '.created_at,.type' events.txt | awk '{ printf("%s", $0); if (NR % 2 == 0) printf("\n"); else printf(","); }'
```

⑫ リポジトリ名を習得

まず、events.txt を jq で整形して、リポジトリ名が `.repo.name` であることを突き止め、ス

クリプトを修正する.

```
jq '.created_at,.repo.name,.type' events.txt | awk '{ printf("%s", $0); if (NR % 3 == 0) printf("\n");  
else printf(","); }'
```

複数の人のデータをまとめたければ, 最初の人だけ > events.csv して新しくファイルを作る.
それ以外の人は, >> events.csv としてファイルに追記するようにする.

⑬ chmod コマンドの使い方

Linux などではシェルスクリプトを書いた時は, 実行可能権をつけていないと, 実行することができない. そんな時に使うのが, chmod コマンドである. その使い方を解説する.

まず, ファイルに対して, どのような権限が存在するのかについて解説する. 基本は以下の3つである.

- r … 読み込み
- w … 書き込み
- x … 実行

そして, その権限を持つ対象として, 以下の4つがある.

u … 所有者

g … グループ

o … その他

a … 上の3つ全て

⑭ 検索

```
python api.py "https://api.github.com/search/repositories?q=mincraft+mod&per_page=100" >  
repositories.txt
```

Minecraft MOD というリポジトリを検索する. ただし, この方法では 1000 件 (100 件 * 10 ページ) までしか結果を取得できない. GitHub 上の設定により最大 1000 件までしか表示されないようになっている.

※gunplot スクリプト解説

表 1 gunplot 解説

言語	解説
set datafile separator ","	データをカンマで区切る.
set xdata time	X 軸が時間であると指定する.
set timefmt "%Y-%m-%d %H:%M:%S"	時間のフォーマットを設定する.
set xl "date time"	X 軸を date time とする.
set yl "number of issues"	Y 軸を number of issues とする.
set terminal png size 1680,1050	結果をサイズ 1680×1050 の.png で出力する.
set border 15 lw 2	外枠全体の太さをサイズ 2 で書く.
set tics font "Times New Roman,15"	目盛の文字を Times New Roman, サイズを 15 にする.
set out "issuesCountChart.png"	issuesCountChart.png という名前で出力する.
plot 'issues.csv' using 1:2 with lines lw 2 title 'open issues', 'issues.csv' using 1:3 with lines lw 2 title 'closed issues'	Issues.csv の 1 列目と 2 列目を使って open issues を描き, issues.csv の 1 列目と 3 列目を使って closed issues を描く.

表 2 コマンド解説

コマンド	解説
curl	プロトコルを用いてデータを通信するコマンドラインツール.
cat	ファイルの内容を閲覧
echo	因数に与えられた文字列を表示する.
>	ファイルに出力する.
>>	ファイルの末尾に追記する
chmod	ファイルやディレクトリのアクセス権を変更する. 600 →所有者にのみ読み込み書き込みの権

	限がある. +x →実行権限を付加する.
-s	行分割だけを行い、行の結合は行わない.
-u	単語間のスペースを 1 つに、文の間はスペース 2 つに減らす.
	コマンドの出力を次のコマンドの入力として渡す.
grep	文字列を検索する.
awk	awk 言語使用の宣言

下の図は上記の調査方法を利用してそれらを自動化し、1 度のコマンド入力でデータの収集ができるようにプログラム概要図を書いたものである.

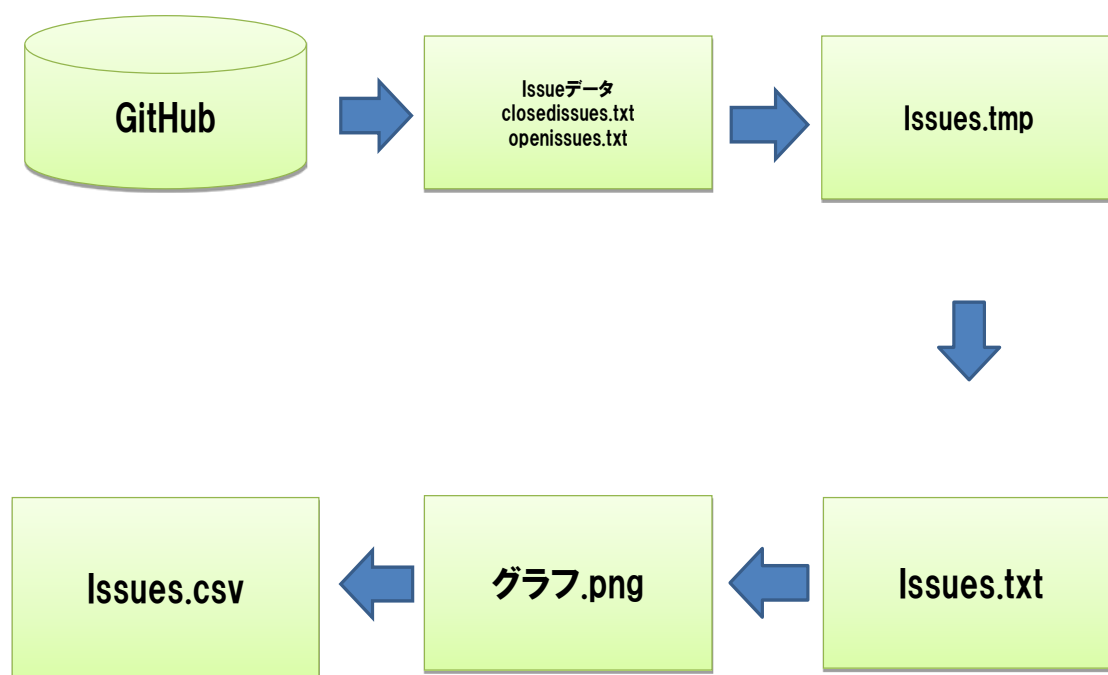


図 12 プログラム概要図

以上を使用し抽出したデータが以下である。

4.5 調査結果

Minecraft のデータだけでは足りないと考え同様に MOD の導入を許可されている skyrim も調査対象としてデータを抽出した。

Skyrim を調査対象として選んだ詳しい理由は、

- Minecraft 同様 MOD が許可されている。
- Minecraft 同様自由度の高いゲーム性である。
- Minecraft の次に MOD 作成人気が高い作品である。

以上の 3 点から skyrim を Minecraft の比較対象として選んだ。

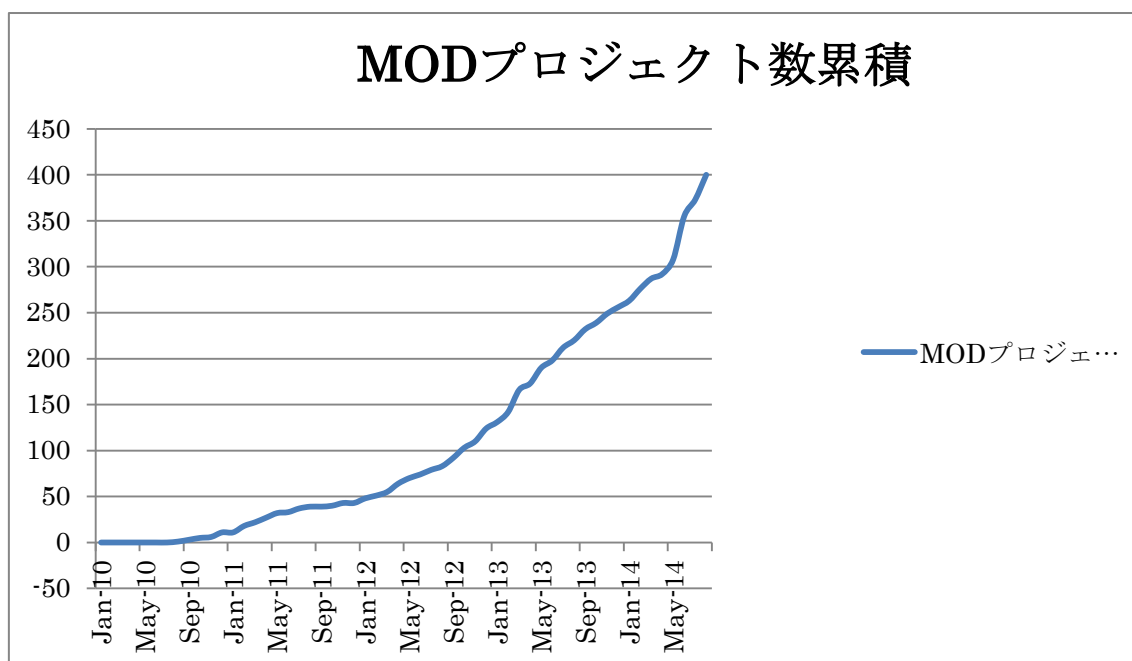


図 13 Minecraft の MOD 数

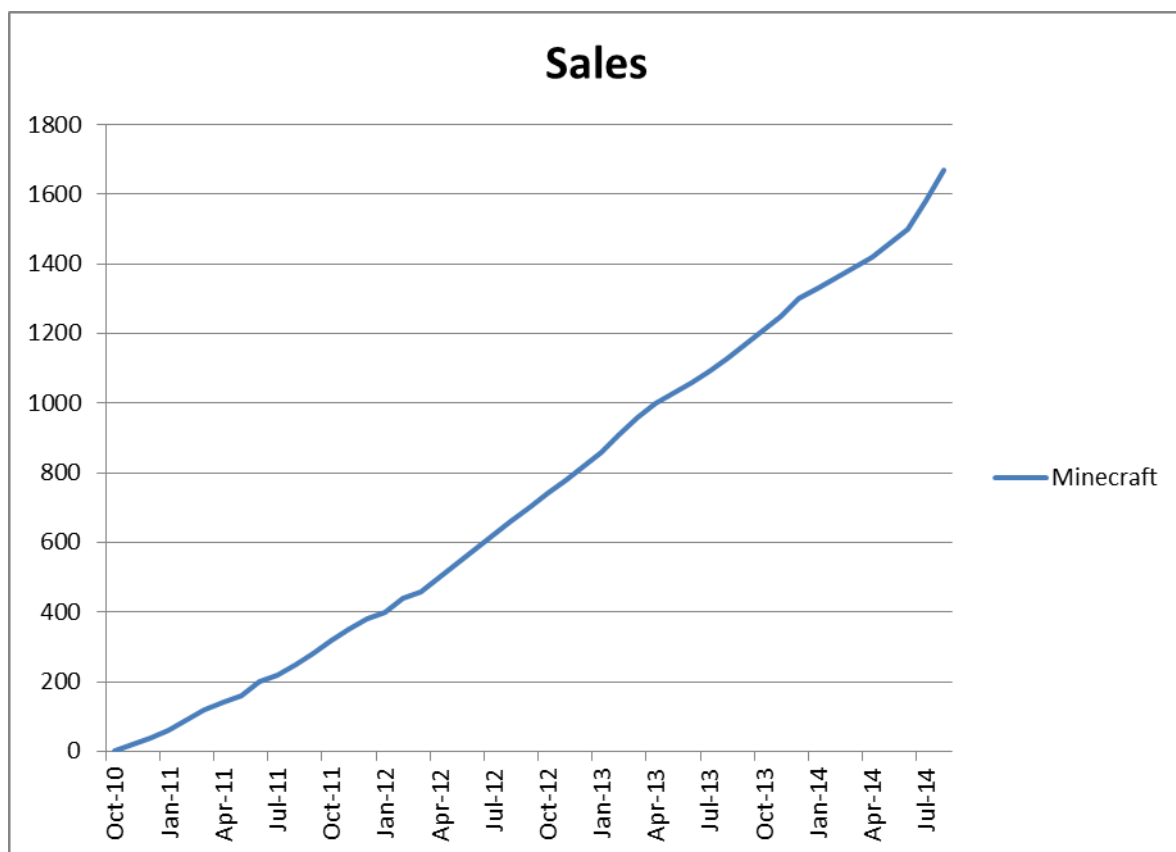


図 14 Minecraft の売上げデータ

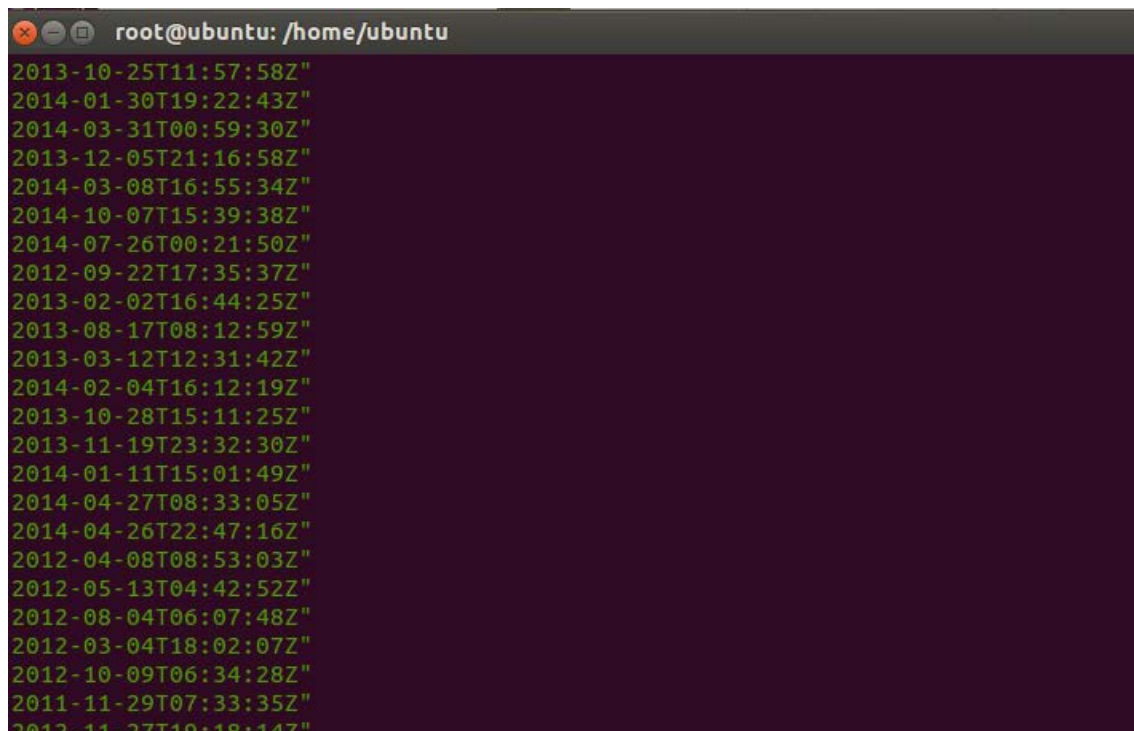


図 15 ubuntu 画面

以上のような結果が出たが、MOD がゲームの売上に関係していると言える結果が出なかった。そのため、その他の要因があると考え、Google Trends を使い Minecraft に関する人気検索ワードを調査することにした。Minecraft が知名度を獲得した理由がトレンドで調査することによって明確になるのではないかと考えたからだ。その結果で売上に関する要因を明確にし、ゲームを販売するときの戦略として確立したいと考える。

4.5.1 Google Trends

Google Trends とは、入力した単語の検索数をグラフで示してくれるサービスである。このときその単語に関連したニュースを見る事ができ、特に検索数が多いときに何があったのかが分かるようになっている。特定の国、期間に絞ったグラフを見ることもでき、また世界中の都市や国、言語毎にどこでの検索が最も多いのかのランキングも提供される。さらにコンマで区切ることにより複数の単語のグラフを比較することも可能である。

以下が Google Trends を使用した結果である。

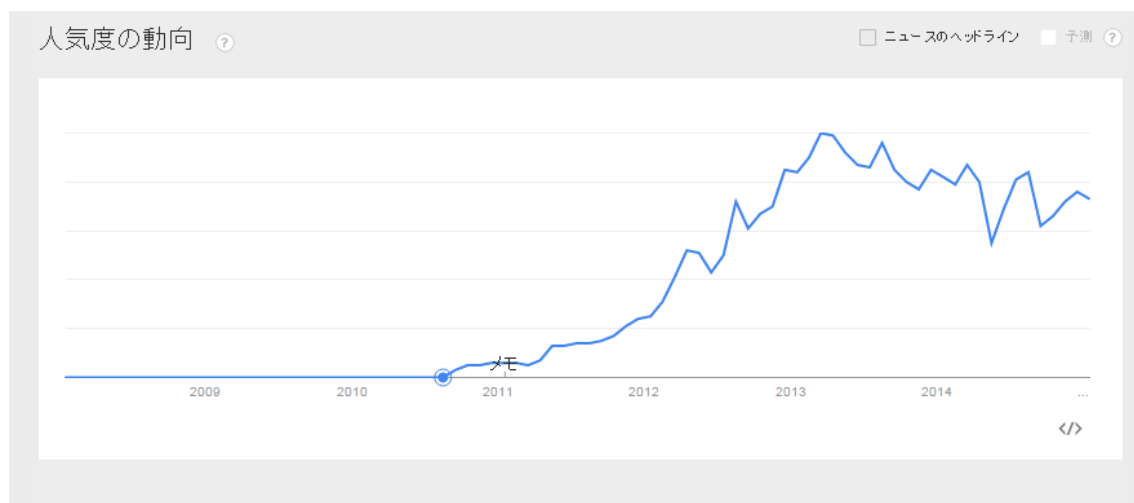


図 16 Google Trends の検索傾向



図 17 Google Trends の人気検索ワード

これを見ると多少のタイムラグはあるが、発売してからすぐに検索量が増えていることがわかる。人気の検索ワードでは「**Minecraft** 実況」と「ゆっくり」というワードが目立つ。これは動画サイト等で **Minecraft** のプレイ動画を実況して投稿していることを指す。

そのため、**Minecraft** が知名度を爆発的に得たきっかけはこの投稿動画に原因があると私は考えた。そこで動画投稿サイトの「ニコニコ動画」についても **Minecraft** と同様に調査対象とすることにした。

4.5.2 ニコニコ動画

ニコニコ動画とは、株式会社ドワンゴが設立し、子会社であるニワンゴが提供している動画共有サービスである。愛称は「ニコ動」や「ニコニコ」がある。事業の拡大につれ、ニコニコ生放送やニコニコ静画など、動画共有サービスの枠を超え、多くの派生サービスが展開されている。新しい総称である **niconico** が発表されて以降、ニコニコ動画は **niconico** のサービスのひとつである動画共有サイトという位置づけとなっている。

ここでは **Minecraft** に関わる動画が1月23日現在89351件存在する。ニコニコ動画に投稿されている **Minecraft** のプレイ動画では、ほぼすべてのユーザーが何らかのMODを使用している状況であることが分かった。中にはMODを紹介する動画も存在する。

以上の調査から考えた私の仮説を図で解説する。

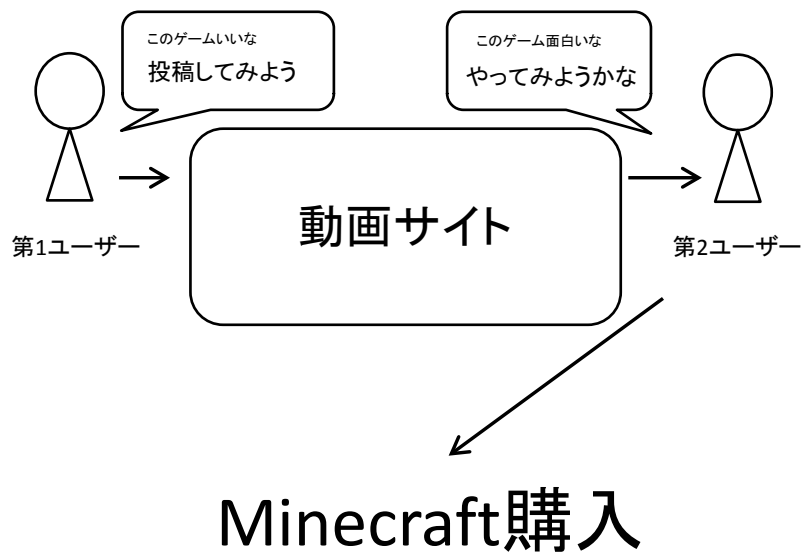


図 18 仮説

4.5.3 仮説

まず、Minecraft を面白いと思ったユーザーや自作した MOD を紹介したいユーザーが動画投稿サイトに動画を投稿する。それを見たユーザーが共感し、実際に Minecraft を購入し MOD を導入する。そしてこの一連の流れが繰り返され今の売上に到達することができたのではないかというものである。これには MOD は直接的には関わっていないが動画を投稿しているほぼすべてのユーザーが MOD を導入していることから MOD を導入することによってゲームの面白みが増し、MOD の存在がなければ成立しないものだと考えても良いだろう。そこから動画サイトを通してユーザーが広告の役割を担っているのではないかと考える。

これを成立させるにはニコニコ動画に投稿されている Minecraft 動画についてのデータを集める必要があるため、公開されているニコニコ API を用いて動画データを収集するツールを作成することにした。

「minecraft」に関連する動画の情報は、次のように取得できる。

```
curl -v "Accept: application/json" -H "Content-type: application/json" -X POST -d
'{"query":"minecraft","service":["video"],"search":["title","description","tags"],"join":["cmsid","title",
,"start_time","view_counter"],"from":0,"size":3,"sort_by":"start_time","issuer":"yourservice/applic
ationname"}' http://api.search.nicovideo.jp/api/snapshot/
```

結果の「total」のところを見ると、動画の数がわかる。

これを書いた時点では 89732 件だったため、次のようなスクリプトですべて取得する（取得中には増えないと仮定している）。

```
rm result.dat

for (( start=0; start<89800; start+=100 ))
do
    curl -v "Accept: application/json" -H "Content-type: application/json" -X POST -d
    '{"query":"minecraft","service":["video"],"search":["title","description","tags"],
    "join":["cmsid","title","start_time","view_counter"],"from":${start},"size":100,"s
    ort_by":"start_time","issuer":"yourservice/applicationname"}'
    http://api.search.nicovideo.jp/api/snapshot/ >> result.dat
done
```

こうしてできる result.dat は JSON のかたまりである。そのためここから必要な情報を抽出する必要がある。

```
cat result.dat | python3 videos.py > minecraft.csv
```

こうして初めて意味のわかるデータ群となる。これをグラフに置き換える。

以下がニコニコ動画における Minecraft に関する動画数の上昇傾向である。

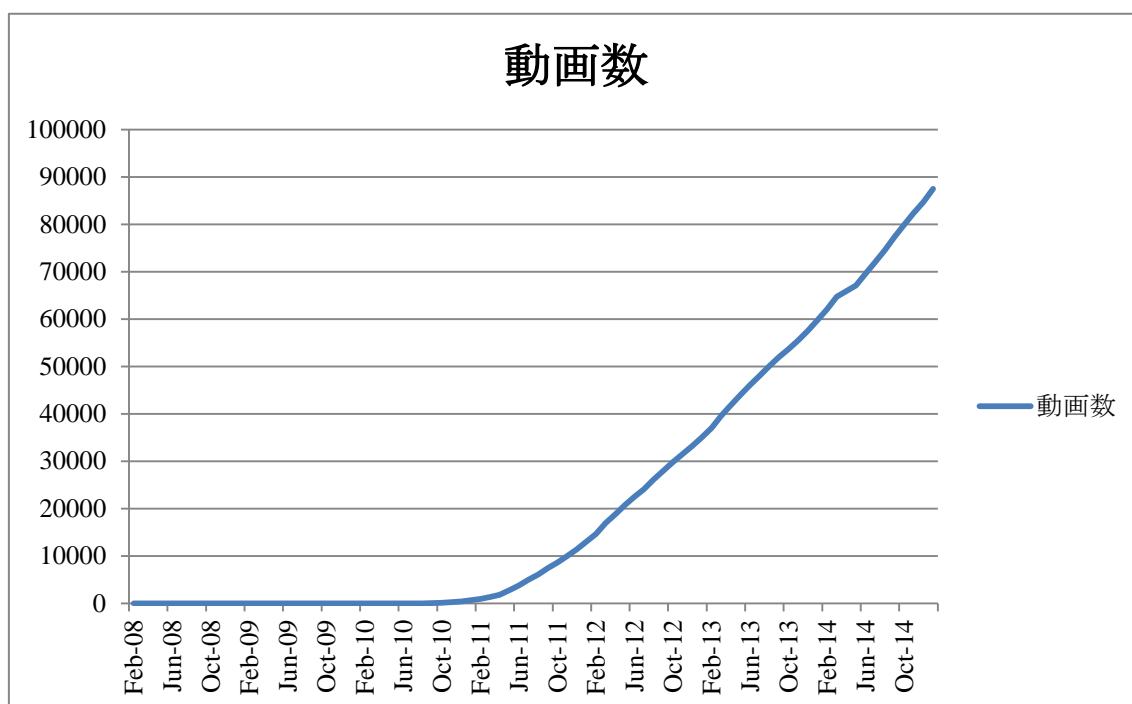


図 19 ニコニコ動画内の Minecraft に関する動画数

この結果から、2011 年 1 月から 2014 年 1 月にかけての上昇傾向は Minecraft の売上の傾向と似ていることがわかる。そのため、一応は Minecraft の売上にプレイ動画の投稿は関係していると言えるのではないだろうか。

参考文献

- [1] Ubuntu Japanese Team. “Ubuntu”. Ubuntu Japanese Team. 2015-1-20.
<https://www.ubuntulinux.jp/home>, (参照 2015-1-20).
- [2] niconico. “ニコニコ動画『スナップショット検索 API』 ガイド”. 2014-10-15.
<http://search.nicovideo.jp/docs/api/snapshot.html>, (参照 2015-01-22).

第五章

考察

5.1 考察

以上の結果から考察を進めていきたいと思う。

Minecraft の売上と MOD 数の変化を 1 つのグラフにまとめた。

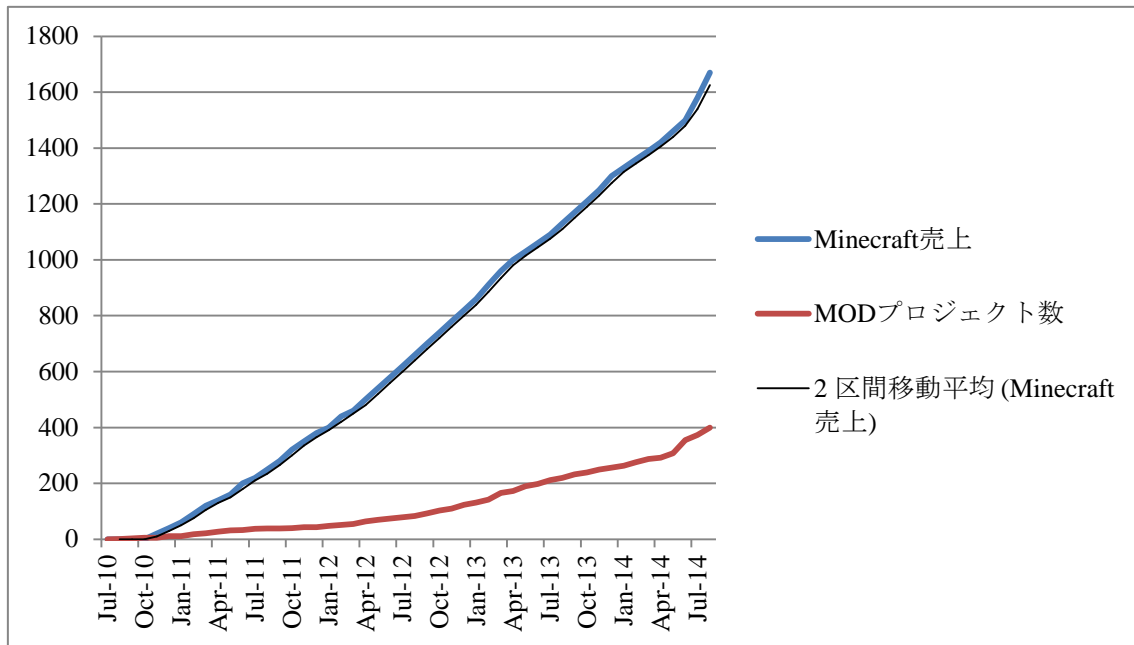


図 20 Minecraft と MOD のグラフ

このグラフを見ても MOD が Minecraft の売上に関係しているとは言えない。売上があるからそれに応じて MOD 製作者が増えたともとれる訳である。

これから考察するにあたって、事業ライフサイクルを参考にゲームにおけるライフサイクルを作る。

本研究の目的を達成したとできるための条件は「Minecraft の MOD プロジェクト数が増えているから Minecraft の売上が上がっている」ということが条件である。

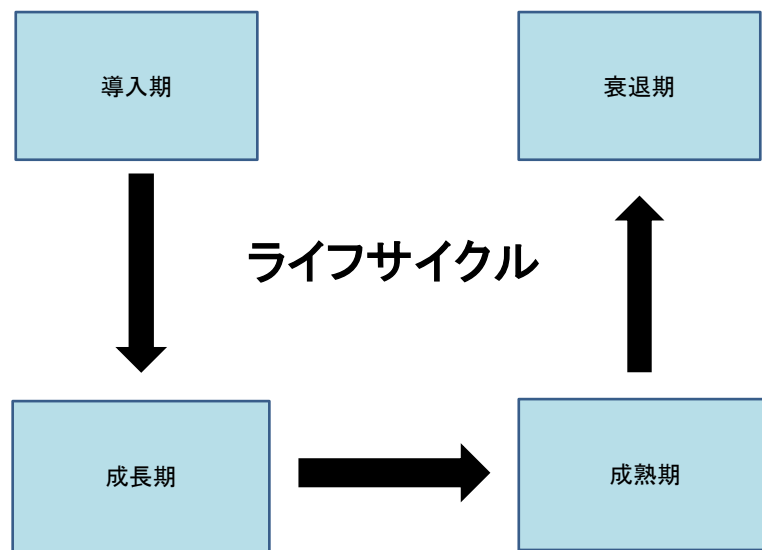


図 21 ライフサイクル

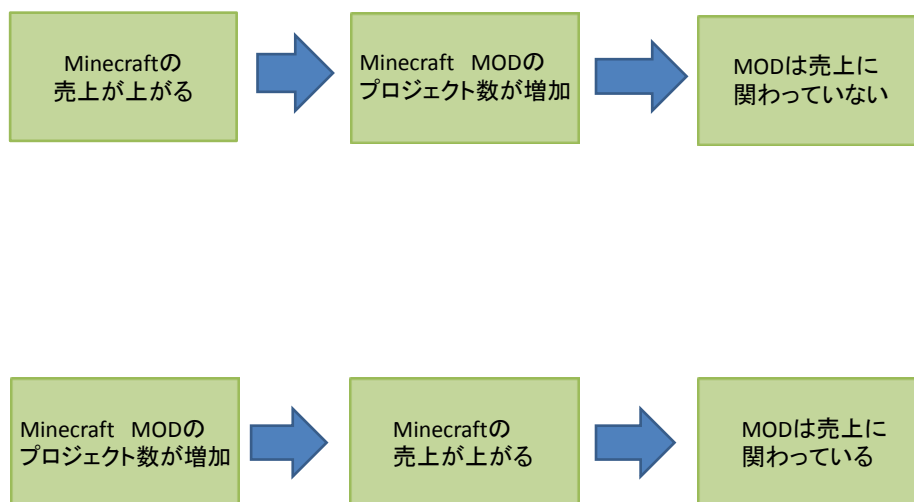


図 22 目的に対する達成条件

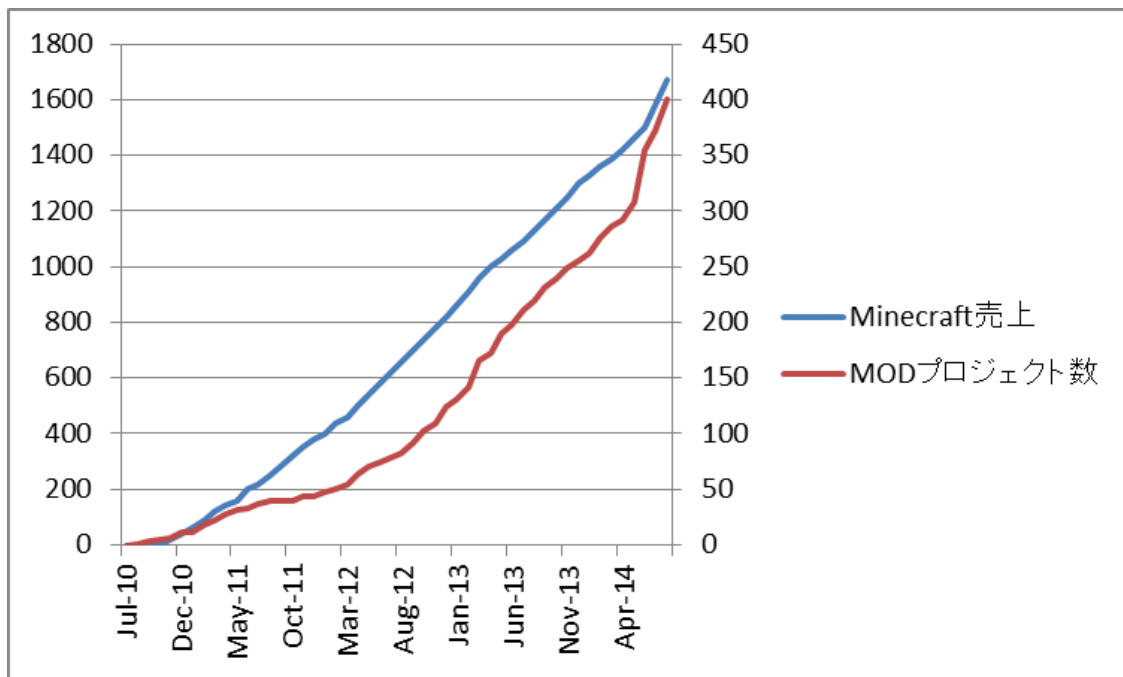


図 23 Minecraft の売上と MOD プロジェクト数の増加幅の比較

上の図から今回の調査結果では、まず先に売上があり人気が出てから MOD 開発が進み売上の増加が緩やかになってから MOD の開発が盛んになったという結果にとれる。

そのため、本研究で求めていた「MOD の開発があるから売上が上がった」とは言えない。成熟期に突入してから衰退期に入るまでの期間を MOD の開発が存在することによって延命できるのではないだろうか。

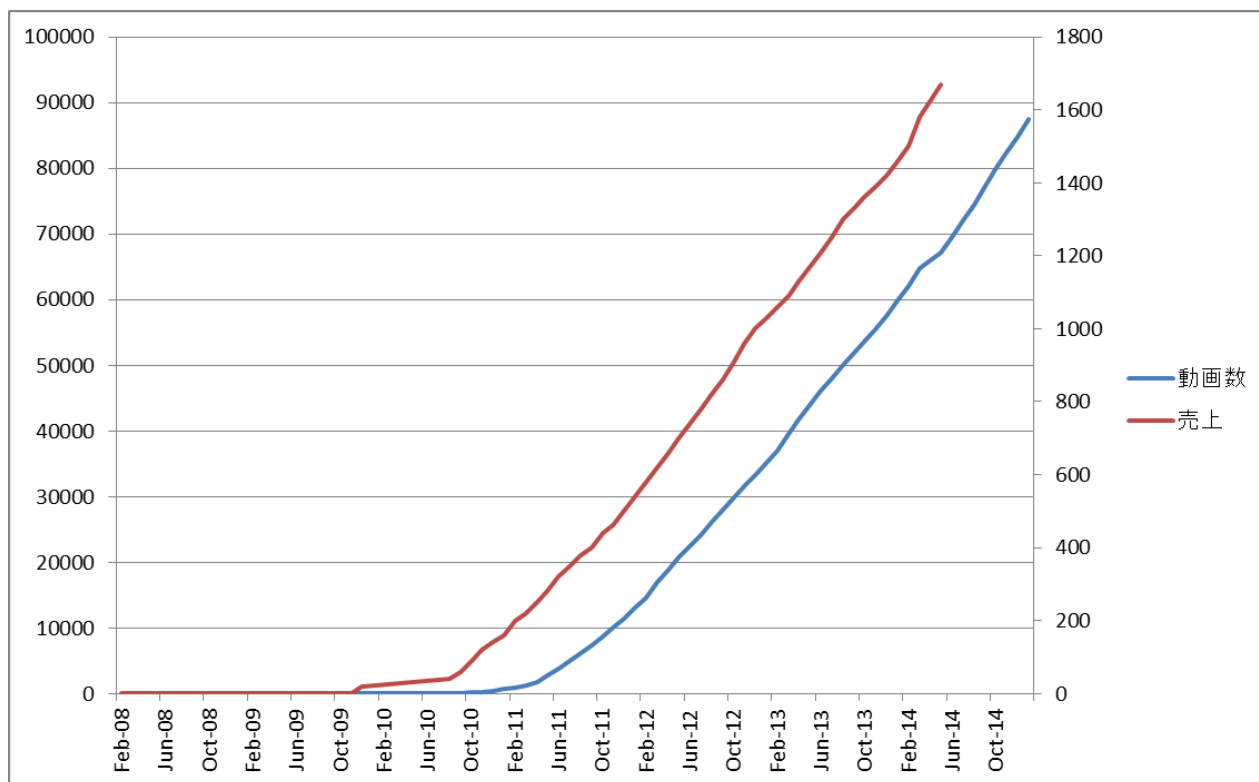


図 24 動画数と Minecraft の売上の増加幅比較

この結果も、ある程度の売上を確保してから動画数が伸び始め、その後の勢いは動画数の増加傾向が上回っていることがわかる。これはある程度 **Minecraft** をプレイするユーザーが増えて、プレイ動画を投稿するユーザーが増えていったと見ていいだろう。この場合ある程度の関連はあると捉えられる。このゲームの成長期における広告効果につながったのではないかと考える。

ここで参考までに各ゲーム機のハード別における売上の上位を見てみる。基本的に動画サイトへの動画投稿は許可しておらず、さらに **MOD** の制作も許可していない作品のため **Minecraft** と比較することができるだろう。

表 3 歴代 PS3 売上上位 10 作品

タイトル	累計	発売日
ファイナルファンタジーXIII	1,905,979	09/12/17
ファイナルファンタジー XIII-2	917,412	11/12/15
バイオハザード6	837,774	12/10/04
ワンピース 海賊無双	828,132	12/03/01
グランド・セフト・オートV	729,533	13/10/10
メタルギア ソリッド 4 ガンズ・オブ・ ザ・パトリオット	706,461	08/06/12
テイルズ オブ エクシリア	670,182	11/09/08
グランツーリスモ5	600,860	10/11/25
龍が如く5 夢、叶えし者	583,549	12/12/06
龍が如く4 伝説を継ぐもの	557,771	10/03/18

表 4 歴代 Wii 売上上位

タイトル	累計	発売日
New スーパーマリオブラザーズ Wii	4,567,937	09/12/03
Wii スポーツ	3,724,264	06/12/02
マリオカートWii	3,701,492	08/04/10
Wii Fit	3,561,787	07/12/01
Wii Sports Resort	3,143,214	09/06/25
はじめてのWii	2,804,201	06/12/02
Wii Fit Plus	2,409,957	09/10/01
Wii Party	2,382,177	10/07/08
大乱闘スマッシュブラザーズX	2,303,103	08/01/31
マリオパーティ8	1,470,332	07/07/26

表 5 歴代 Wii U 売上上位

タイトル	累計	発売日
New スーパーマリオブラザーズ U	1,203,291	12/12/08
マリオカート 8	896,532	14/05/29
W i i P a r t y U	781,872	13/10/31
大乱闘スマッシュブラザーズ for Wii U	581,791	14/12/06
スーパーマリオ 3D ワールド	573,542	13/11/21
N i n t e n d o L a n d	382,954	12/12/08
モンスターハンター 3 G H D Ver.	233,810	12/12/08
ピクミン 3	232,313	13/07/13
W i i F i t U (本体同梱版先行発売)	176,265	13/10/31,14/2/1
太鼓の達人 Wii U ぱ〜じょん！	162,440	13/11/21

この累積の売上をみるに、Minecraft の 1600 万ダウンロードという数字はすさまじいものであるということがわかる。時系列グラフは手に入れることができなかったが、発売日からの累積値を見ても何か爆発的にヒットする要因が Minecraft にはあったのだということがわかる。

5.1.1 ライフサイクル

本来、ゲームは販売しようとしたとき開発費以外に初回の広告費が必要となる（導入期）。その広告効果に応じて製品の売上に関わってくる。その後、人の口コミやレビュー等で自動的に拡散され成長期へと至る。ある程度成長仕切った後、成熟期とされ比較的売上の増加量は緩やかになっていく。その後、衰退期のフェーズに突入しゲームをプレイするユーザーの勢いは衰え、売上はこれ以上伸びないようになる。

これが図 5-2 に記した通り、ゲームにおけるライフサイクルである。

しかし、今回の調査結果から既存のライフサイクルよりもより付加価値のあるライフサイクルの形をとった **Minecraft** のライフサイクルを解説したいと思う。

① 導入期

製品に付加価値を付けて売りに出す。新しい試みだったり、性能だったり、とにかく人の目につく様なモノが前提として必要である。その中で広告費を導入してみたり、営業に行ってみたりと何かと行動することが重要である。**Minecraft** はその中でサンドボックスゲームとクラフティングの組み合わせという試みお行い、それが見事に成功した。さらに拡張要素の **MOD** の導入を許可したり、動画投稿サイトにプレイ動画の投稿を許可したりするなど工夫を加えた。

② 成長期

導入期である程度ユーザーを獲得したことで、広告費等をあまりかけずとも自然と拡散して売上に繋がるようになる。さらに動画投稿サイト等でプレイ動画のアップロードを許可していたり、**MOD** の制作を許可していたりしたことにより、従来のライフサイクルよりも効果的な結果が得られた可能性が高い。

③ 成熟期

成長期よりも売上の増加傾向は緩やかになりはじめる。従来のライフサイクルではこのまま衰退期に突入し終わりを待つだけになるが、**Minecraft** では **MOD** の導入を許可したことにより衰退期への突入を先延ばしにできた。その理由としてグラフにも現れている通り、一度緩やかになってから再度緩やかに上昇しているためである。それと同時期に **MOD** のプロジェクト数が増加しているためその時期に **MOD** 導入を許可していたことに対する結果がでていると考えられる。

④ 衰退期

まだ Minecraft では衰退期に突入しているという印象は受けないがどのコンテンツも必ず終わりへ向かうので、Minecraft もいつか終わりが来るだろう。

極端な例ではあるが、Minecraft の様なライフサイクルで発売したゲームと従来のライフサイクルで発売したゲームの売上の比較をする。

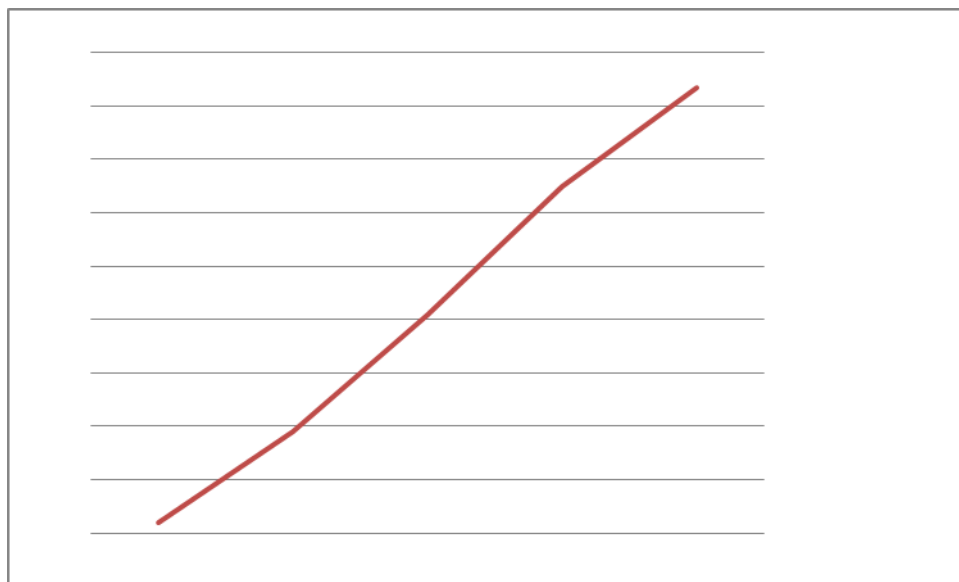


図 25 新しいライフサイクルの売上の形

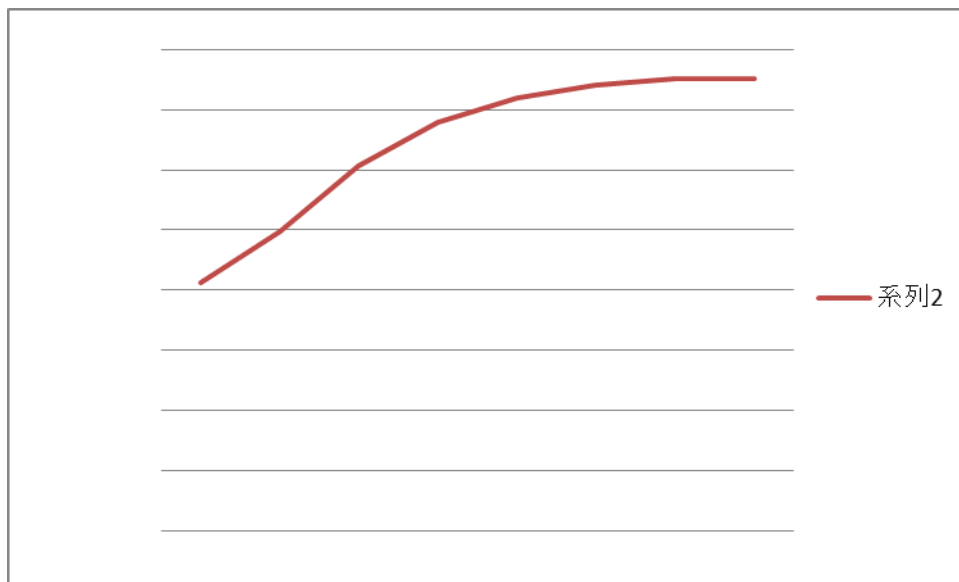


図 26 従来のライフサイクルにおける売上の形

この様に、従来のライフサイクルでの発売では導入期と成長期にかけて広告費を大量に投入し、最初の売上がある程度確保されてから緩やかに衰えている。しかし、Minecraft

のような自然にユーザー間の口コミで人気を獲得してきたゲームは売上の伸びは若干衰えてはいるものの、今も尚売れるゲームとなっている。

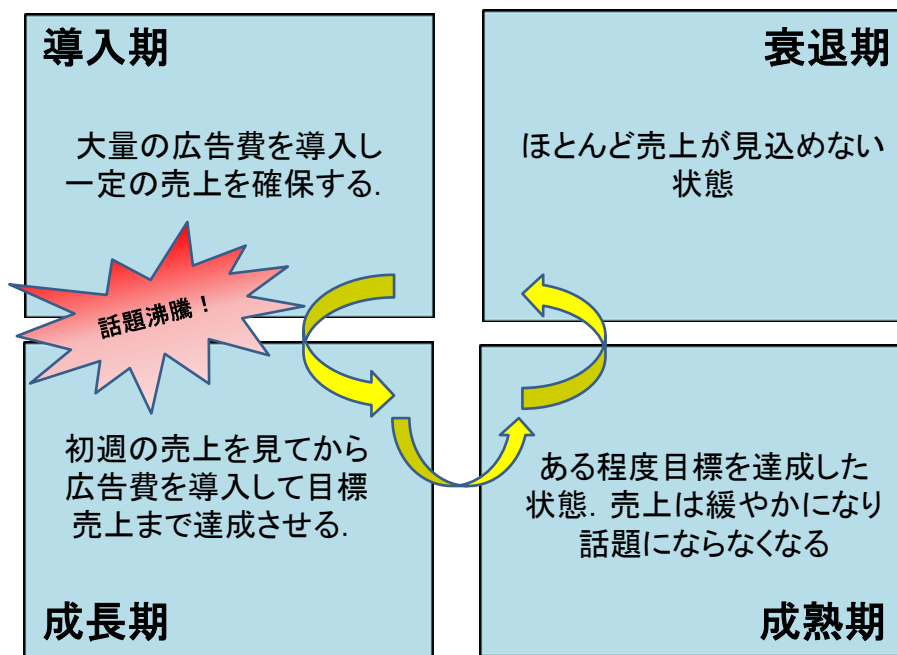


図 27 従来のライフサイクル図

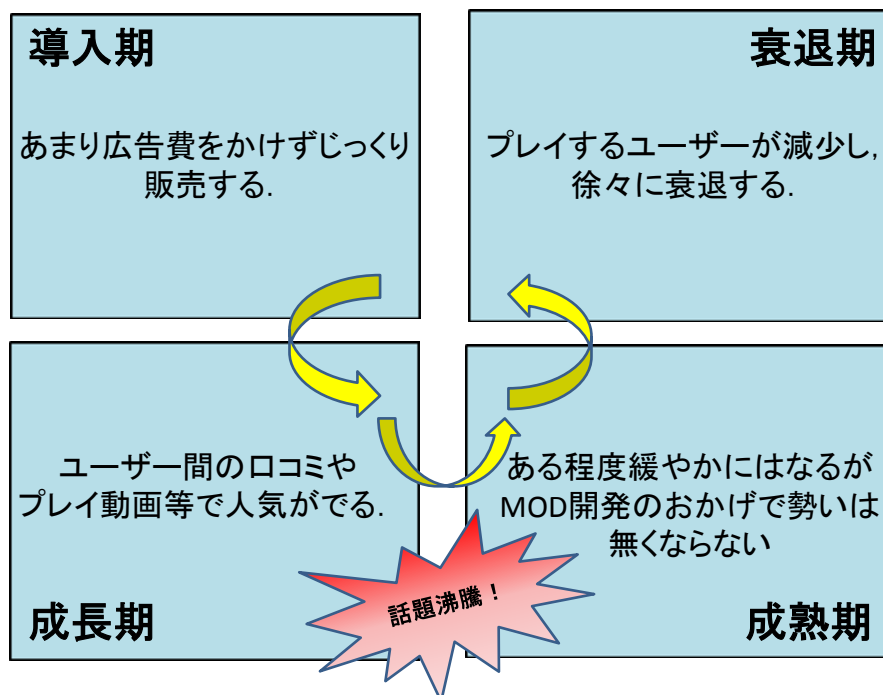


図 28Minecraft のライフサイクル図

既存のライフサイクルのメリットはある程度広告費を導入する分、どのくらい売れるか等の予想がたてやすい。シリーズ作品のナンバリングタイトルや目玉タイトルを売り出す時に役に立つと考える。しかし、デメリットとして、広告費がかかり過ぎるという点が挙げられる。そのため、新規タイトルや新しい試みをする際には、売上が伸びなかった場合におけるリスクが高くなってくる。

そして、新しいライフサイクルのメリットは、既存のライフサイクルとは逆にユーザーのロコミや、ユーザーのプレイ動画投稿等を広告のような役割とさせているため広告費をかけることなく販売できる。さらに、拡張要素を取り入れているため既存のライフサイクルよりも長い期間で収益をあげることができる。デメリットとして導入期の時点で付加価値を付けられなかった場合そもそもの売上が期待できない可能性があること。集客性が足りず成長期にうまく伸びずそのまま衰退してしまう可能性があることが挙げられる。そのため、ナンバリングタイトルである程度稼げるようなゲームでこれを行うには少しリスクが高すぎる。新しいゲーム性のものを世に出したい場合やそもそもあまり収益にこだわっていない場合などにこの手法は最適なのかかもしれない。

5.2 研究の反省・まとめ

本研究での目的はMODが売上に関係しているのかを明確にすることであったが、提案したライフサイクルも完全に正しいとは言えない結果であり、それはデータサンプルの不足からきている。MODやプレイ動画を投稿することを許可して尚且つその選択は成功していると考えられるサンプルがMinecraftしかなかったからだ。そのため実際にやってみる他ないと考えられる。売上が上がったから動画数も増加しMODプロジェクト数も増加したと言えるし、動画数やMODプロジェクト数が増加したから売上也増加したととれるのだ。そのためこの研究はまだ不十分であると私は考えている。しかし、今後何かを販売する際に戦略を立てようと考えた時、自動でデータを収集できるツールを制作したためこれを今後に役立てていきたいと考えている。

5.3 謝辞

本論文を作成するにあたり、指導教官矢吹太郎准教授から多くの指導をうりました。ここに感謝の意を表します。

参考文献

[1]VGChartz. Game DB, 2015/01/20 <http://www.vgchartz.com/>, (参照 2014/01/20).