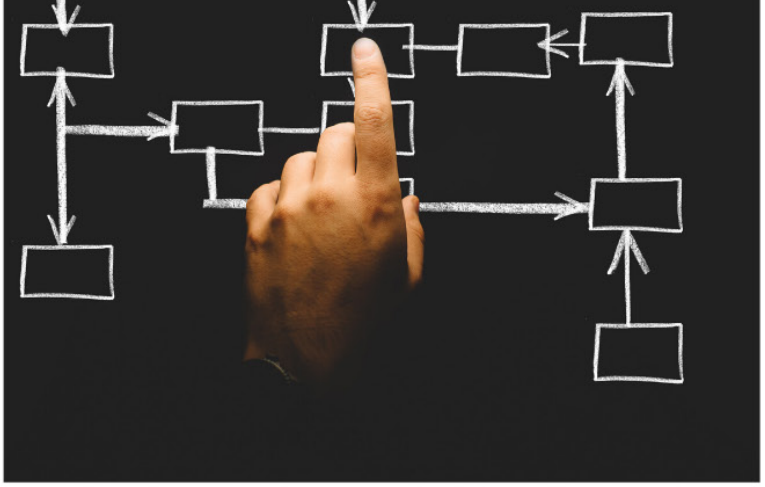

一瞬で全体像を掴む

ゲーム開発の攻略チャート

TODO（エクスプラボ）

目次

設計

準備

開発

テスト

リリース

広報

終わりに

はじめに

この『一瞬で全体像を掴む　ゲーム開発の攻略チャート』は、Unity を使ってゲーム開発をしたいと思っている方向けに書かれたレポートです。

todo がスマートフォン向けのゲームアプリをリリースした時の流れをベースに、あなたがゲームを作成する際のガイドとなるようにまとめました。ゲーム作りはそれ自体がゲームのような楽しさを持っているので、攻略本のような位置付けとなるよう『攻略チャート』と名付けています。

作成するゲームのジャンルや規模によっては考慮しなくて良い項目もあるため、必要な部分を抜き出してオリジナルのチャートを作ると便利です。

このガイドはあくまで目安であり、このガイドに沿ってゲームを作成したことで必ず成功することまではお約束できません。なぜなら、あなたのゲームが成功するかどうかは、ユーザに楽しんでもらうためのあなたのアイディアや、ゲームを完成させるためのあなたの行動にかかっているからです。

しかし、ゲーム作成の道筋を示すことであなたの迷いを取り払い、完成までのプロセスを加速させることはお約束できます。

もし「ゲームを作るために何をしたらいいか分からない」と迷ったことがあれば、このガイドはきっとあなたのお役に立てます。

ゲーム開発の流れ

ゲームを開発する時、まずはじめに行うのが「**企画**」です。
あなたの頭の中にあるゲームの姿を少しずつ言葉にして形を与えていきます。このゲームでユーザがどのように楽しんでくれるのかを想像しながら、アイディアをまとめていきましょう。

企画がまとまってきたら「**設計**」に入ります。
アイディアをより具体的な振る舞いとして考え、動きを決めていきます。この振る舞いを決めた後に、どのようにプログラミングをしていくか決めるとスムーズです。

設計が終わったら開発のための「**準備**」を行います。
Unity のインストールや、ライセンスの準備、外部サーバの準備など、開発を行うために必要な作業を終わらせましょう。

準備が整ったらいよいよ「**開発**」です。
設計で決めた内容にしたがって設定やプログラミングを行います。実際に開発している中ではある程度の設計変更が出てくることがあるため、柔軟に対応します。

ゲームが出来上がってきたら「**テスト**」を行います。
設計で意図した通りにゲームが動いているか、企画で想定したユーザが楽しんでいるポイントを表現できているか、ゲームのバランスは問題ないかといった観点からゲームをテストします。テストこそ開発の中で一番大切なフェーズです。意図したことが表現できていなければ、良い企画であったとしてもユーザに伝わらないことになってしまいます。この点は力を入れて頑張りましょう。

テストが終わったら「**リリース**」を行います。
作ったゲームは誰かに遊んでもらってこそ。ユーザに届けるために確認をしっかり行なっていきます。

リリースと並行して「**広報**」も行います。
アプリストアに掲載しただけでは、多くの人はあなたのゲームに気付きません。世の中にある様々な商品が「私を見て！」と宣伝しているのと同じように、あなたのゲームに気付いてもらうために宣伝を行いましょう。宣伝を行なってユーザの反応をもらえたら、それを元にアプリのアップデートを行い、さらに多くのユーザに届けていきましょう。

企画

作成したいゲームの内容を考えていきます。
このフェーズでは「どんなゲームを作りたいか」を中心にざっくりとゲームのアイディアをまとめます。ゲームを遊んでくれるユーザがどのような要素に熱中してくれるのか、といった視点も忘れずに。
この段階でリリースするプラットフォーム、国や地域、販売価格なども決めておきます。

ゲームの内容を決める

あなたの頭の中にあるゲームの姿を形にしていきます。
どのようなジャンルのゲームか、ユーザはどんな点で面白さを感じるのか、紙に書いたり PC で打ち込んでいきます。
もし数人で話し合いながらアイディアを出しているのであれば、ホワイトボードにどんどん書き込んでいくのもいいですね。

ゲームの名前

まずはゲームに名前を付けましょう。
ここで決める名前は仮置きのもので大丈夫です。プロジェクト期間中に呼ぶコードネームでもいいかもしれません。名前があると自分で開発するときもチームで開発するときも識別しやすくなります。

ジャンル

ゲームのジャンルを決めます。
アクション、RPG、レーシング、アドベンチャー、スポーツ、格闘、リズム、パズル、ボードゲームなどなど、あなたのゲームを表すのに分かりやすいジャンルを決めましょう。いくつかのジャンルを組み合わせても良いと思います。

ゲームの要素

ゲームの要素を決めます。
ゲームの要素はプレイヤーがゲームを遊ぶときに中心的に楽しむポイントです。
例えば RPG なら探索と戦闘、レーシングゲームであればライバルとの競争といったように、ゲームのキーとなる要素を書き出してみましょう。

テーマ

ゲーム全体をまとめるテーマを決めます。
ファンタジーだったり、SF の世界だったりと、ゲーム内の要素が統一感を持っているようにするとプレイヤーとしてもゲームの世界に入っていきやすいですよね。ゲームボーイのサガシリーズのように、関羽の鎧を装備したエスパーがサブマシンガン片手に魔法を唱えるのもあり。「そういう世界だ」とプレイヤーに説得力を持って伝えられれば OK です。
でも最初は分かりやすくまとめた方が楽かも。

雰囲気

テーマと関連しているポイント。
決めたテーマの内容をどう表現するかの観点です。
ファンタジーなら絵本のような雰囲気だったり、レーシングならリアルな車が競い合う雰囲気だったり。ホラーゲームでもリアルっぽくゾンビを表現するのと、初期プレステのようなポリゴンで表現するのでは印象が変わります。

2D か 3D か

Unity でゲームを作る上で 2D のゲームにするか 3D のゲームにするかは大切なポイントです。
3D であれば 3D モデリングやライティングを駆使して広い世界を表現できますが、一方で 2D のゲームは携帯機で遊びやすい側面もあります。
2019 年現在、新しく発売されるスマートフォンの性能はどんどん上がっているため、3D でも耐え得るスペックのものもあります。ただ古い端末を使い続ける人 (ええ、私です) もいるため、あなたがゲームで表現したい世界とプラットフォームのスペックのバランスを取るのも大切です。
ここは設計に大きく関わる部分なので、企画の段階で決めてしまうと後の作業がやりやすくなります。個人的に最初は 2D が作りやすくていいと感じています。

プレイ人数

ゲームを遊ぶ人数を決めます。
スマートフォンアプリであれば基本的に一人プレイを想定しておけば良いかと思います。
サーバを介して同時に対戦するゲームも作れるので、そうした場合は同時プレイの人数を考えておきましょう。
家庭用ゲーム機などを対象にしている場合は、遊んでいる人数に応じて CPU などの導入を検討する必要があるかもしれません。

参考にしたゲーム

あなたがゲームを思い描くにあたって参考にしたゲームがあれば書き留めておきます。
チームで開発しているときは「ドラクエっぽいゲーム」とか「FF っぽいゲーム」といった形で仲間に伝えるとイメージしやすくなります。
自分一人で開発している場合にも完成形がイメージしやすくなるメリットがあります。
もちろん、あまりに意識しすぎると同じものを作ってしまうので、自分の色を大切に。

プレイヤーが熱中する要素

あなたのゲームを遊んだプレイヤーはどんな要素に熱中するのか、プレイヤーの視点から想像するといいでしょう。
RPG だったら宝を集めたり、強大な敵を倒したりすることが熱中する要素かもしれません。
アドベンチャーゲームだったら謎解きやストーリなどかもしれません。
まずあなた自身があなたのゲームを頭の中で遊んでみて、「ここはプレイヤーに楽しんでもらえそうだ」「この部分は人に話したくなる」と思えるポイントを書き出してみましょう。

プログラミング言語

ゲームの機能を実装するプログラミング言語を決めます。

クライアント側 (ソフトウェア、アプリ)

プレイヤーの手元に配布されるソフトウェア (アプリ含む) のプログラミング言語を決めます。
Unity を使っている場合は C# になるかと思います。

サーバサイド

ソフトウェアやアプリと通信するサーバを用意する場合は、サーバ側の処理を記述するプログラミング言語を決めます。
Ruby、PHP、Python、JavaScript などの軽量プログラミング言語だったり、コンパイルが必要な Java や C# を使う人もいます。アプリの規模 (リクエストの数や負荷など) に応じて適切な言語を選んでください。
正直なところ一から作るとなると大変なので、バックエンドサービスなどを使うと素早く開発できます。その場合もクライアント側とのデータのやり取りをする部分では JavaScript で記述する必要があったりと、多少のコーディングが必要になります。

リリースするプラットフォーム

どのプラットフォームでリリースするかを決めます。
家庭用ゲーム機、スマートフォン、PC 向けなど、ゲームを遊べる環境は年々増えています。あなたの思い描いたゲームの世界を表現できるプラットフォームを選択しましょう。また、プラットフォームごとに機能的な制約などもあるため、この辺りは設計に関わってくる部分となっています。

デバイス

ゲームをプレイする端末を決めます。
家庭用ゲーム機を対象とするのであれば、Nintendo Switch、PS4 などがあります。個人でこれらのプラットフォームにゲームをリリースするのはハードルが高いので、パブリッシャーさんと組む必要があるかもしれません。
PC 向けであれば Steam でリリースして、Windows の PC で遊んでもらうことになります。その際、ゲーム用コントローラがあった方が良いのか、キーボードを使うのか、といった部分も考えておくと良いでしょう。
スマートフォンを対象にする場合は、iOS の端末は iPhone や iPad があり、Android も各種スマートフォンに加えてタブレット端末があります。
タブレット端末ではスマートフォンと画面の比率が異なるため、対象に含めるかどうかで設計時に考慮すべき点が変わります。

リリース日

ゲームをリリースする日を決めます。
まずは目標とする日付で大丈夫です。
Unity でゲームを作るのが初めての場合は、最初に目標とした期間に対して 1.5 倍から 2 倍の時間を見込んだほうが良いです。
特に今会社で働いている人であれば仕事のない日に開発するのが中心となるかと思うので、プライベートでも無茶な納期で精神を削るよりは、余裕のあるスケジュールで計画的に進める方が精神的にも楽です。
もちろん、開発の状況によって前後することはあります。
この場合、リリース後の振り返りで当初の予定と実績とで比較を行い、その原因・理由を考えておくことで次回作の開発に役立てることができます。

リリースする国、地域

ゲームをリリースする国や地域を決めます。
最初はあなたがいる国を対象としてリリースするのがおすすめです。
アプリを多言語化するといっても、単純に文章を翻訳すればいいのではなく、その国の人の感覚・感性に合わせたゲーム作りが必要になります。これがあるため多言語化にかかるコストは無視できないレベルとなってしまいます。
最初のアプリでこれをやると 8 割の人は計画を頓挫して諦めてしまうので、まずはあなたのゲームをあなたの国のプレイヤーに遊んでもらうことを優先するのがおすすめです。

マネタイズ

マネタイズはあなたのゲームによる収益化のことです。
広告収入を得るのか、アプリ自体を有料で販売するのか、基本無料でゲーム内課金を行うのか。
収益化の方法はいくつかあるのであなたに合った方法を選択しましょう。
もちろん、広告を導入しないで完全に無料にすることも可能です。

広告の有無

ゲーム内で広告を表示するかどうかを決めます。
Unity の場合は Unity Ads という広告 SDK が用意されているので簡単に導入することができます。広告を見るとゲーム内でメリットが得られるリワード広告を使うと広告を見てもらいやすくなるかもしれません。

販売価格

ゲームの販売価格を決めます。スマートフォンアプリであれば、アプリストアでダウンロードする際の販売価格を決めます。個人的には最初は無料アプリとしてリリースした方が遊んでもらいやすいと思っています。
ゲームが売れてお金が入ってくるというのはモチベーションに繋がるので、ある程度ゲーム作りに慣れてきたら有料で販売することも検討してみてください。

設計

企画した内容を設計に落とし込んでいきます。
ゲームでプレイヤーが何をして、その結果どうなるのか、といった振る舞いを決めていきます。
仕事で作る設計書のようにきっちりとしたフォーマットでなくても良く、ノートなどに書いても良いです。
開発期間中に何度も見るものなので、文章と図を駆使して分かりやすくまとめておきましょう。
また、設計を固めるにあたっては Unity の画面を見ながら、あるいは簡単にパーツを作りながら進めた方が早い場合もあります。

ゲームのストーリー

RPG やアドベンチャーゲーム、ノベルゲームではストーリーがゲームの核となります。
しかし他のゲームであっても、プレイヤーがゲームの世界に入っていくためにはストーリーが重要です。
将棋のゲームやパズルゲームにストーリーなんて……と思われるかもしれませんが、「あなたの」ゲームとして他と差別化できるのはこの部分です。
ただ将棋ができるゲームと、主人公が奨励会に入ってタイトルを獲るべく、ライバルたちと切磋琢磨しながら将棋をするゲームでは印象が変わります。
ただ単に将棋ができるゲームをしたいユーザは「あなたの」ゲームを遊びたいのではなく、単純に将棋がしたいだけです。
確かプロのクリエイターの方で「大切なのは作家性」とおっしゃっている方がいました。
テーマや雰囲気と合わせて、あなたらしさの滲み出るゲームを遊んでもらえるようなストーリーがあるのが理想ですね。

ゲームのルール

ゲームのルールを決めます。
ゲームの始まり、何をしたら終わるのかといった目的、その過程でプレイヤーができることは何かを決めていきます。ゲームオーバーの条件、ステージクリアの条件、キャラクターの成長などもここに含まれます。

RPG なら戦闘システムの設計もここでやっておきましょう。
出発点は企画段階で決めたゲームのジャンルで一般的なルールに合わせ、独自のルールが必要であればそこに追加していきます。
ゲームに慣れているプレイヤーであればルールの把握が早いですが、慣れていないプレイヤーは慣れるのに時間がかかります。
独自ルールが多くなりすぎると説明も必要になってくるので、「ちょっとだけ新しい」くらいのルールにするとバランスが良いと思います。

登場するキャラクター

ストーリーと関連してゲームに登場するキャラクターを決めます。
分かりやすいのは主人公と敵キャラでしょうか。
ストーリーに応じて仲間のキャラクターが何人かいたり、敵の陣営にも複数のキャラクターがいるかもしれません。キャラクター同士の相関図があるとストーリーも作りやすいです。
ストーリーがほとんど必要ない場合は、主人公がプレイヤー、敵あるいはガイドがゲームマスター (あなた) になるかもしれません。

ゲームの操作方法

プレイヤーがゲーム内で行う操作を決めます。
プレイヤーが主人公となるキャラクターを操作して画面内を動かすのか、それとも画面のボタンをタップしてゲームを進めていくのか、こうした部分を詰めていきます。
キャラクターを動かす場合はコントローラの動作をここで決めます。
スマホアプリであれば画面内に UI でコントローラ用のボタンを表示する必要があります。というのも、スマホには物理的なコントローラがないためです。
キャラクターが画面内を動き回るのではなくボタンをタップして進めていく場合はそれぞれのボタンの動きを決めていきます。
ボタンを画面内に表示するため、操作の数が多くなるとプレイヤーが混乱する原因にもなります。なるべく簡単に操作できるのが望ましいですね。

必要な画面

ゲームで必要な画面を決めていきます。
タイトル画面、オープニング画面、ステージセレクト画面、ステージの画面、リザルト画面、といったように役割ごとに画面を分けましょう。
ステージ数についてもここで決めます。
このタイミングで画面のイメージを絵で描いておくと次のシーン設計や UI の実装方式の作業がやり易くなります。絵は紙に手書きすれば十分です。
大切なのは画面内に必要な要素をイメージすること、大体の配置をイメージできることです。
たとえ線が歪んでいても Unity 内で数字で指定すれば揃うので、ささっと頭の中にある画面を書き出してみましょう。

シーン

必要な画面を決めたら、機能ごとにシーンを分割します。
シーン同士の遷移でフェードイン / フェードアウトなどの演出を使う場合はその表現方法も決めておきます。

必要なシーンの数

上で決めた必要な画面から、Unity で作成する必要のあるシーン数を決めます。
同じ機能は同じシーンでまとめると開発していて分かりやすいですし、1 シーンごとのメモリ使用量も抑えられます。
特に 3D のゲームでは画面内にたくさんのオブジェクトを描画する必要があるので、それに従ってメモリ使用量が大きくなりがち。
スマホ向けのゲームであれば、なるべくパフォーマンス面の負荷を抑えられるようにシーンの大きさにも気を配りたいところです。

シーン遷移の表現

別のシーンを読み込む処理を実装する場合には、フェードイン / フェードアウトなどの演出を入れると自然な遷移になります。
画面遷移時の演出は「トランジション」と呼ばれ、フェードの他にもワイプ、スライドといったように様々な表現方法があります。
個人的にはブラックアウト (暗転) を使って画面を真っ暗にして、その間にシーンをロードすることが多いです。真っ暗な画面に「Now Loading」と表示すれば、プレイヤーから見ても「あ、今ロード中なんだな」と分かりやすいですからね。

カメラの動作

シーン内でカメラを動かすかどうかを決めます。
画面内をキャラクターが動き回るゲームであればカメラで追いかけることが多いでしょうし、ステージが画面に収まるのであれば特にカメラを動かす必要はないでしょう。
演出としてカメラをズームイン / アウトしたり、パンで横に振ったりすることもあるかもしれません。
シーンの役割とゲームの演出方法から決めると良いでしょう。

(2D) タイルマップの使用

2D のゲームではステージを表現するために Unity のタイルマップの機能を使うことができます。
マップチップから自由にステージを作成できるため、どのようなステージを作るかを決めておくと作業が早く進みます。
人によっては別のツールを使って 2D のステージを作り、画像としてインポートした方が早い場合もあるので、その点も考慮しながら使うか使わないかを決めると良いと思います。

Rigidbody や Collider の使用

物理的な運動をさせるか、接触判定は必要か、といった観点から Rigidbody や Collider を使用するかどうか決めます。
どちらも 2D、3D で使えるので活用すると便利です。
Collider は敵との接触を検知したり、アイテムを拾う判定をしたりと、特に使う機会が多いかもしれません。

UI の実装方式

必要な画面で決めた画面イメージを元に、必要な UI パーツを決めていきます。画像の規格やテキストの規格を決めてあらかじめ揃えておくと開発フェーズで実装しやすくなります。

Canvas の規格

UI を描画するキャンバスの大きさを決めます。画像サイズを決めるにあたって先に決めておくと楽です。

サイズ

Canvas のサイズを決めます。家庭用ゲーム機や多くのスマホでは画面の横と縦の比率が 16:9 に近いので、これをベースに決めると良いでしょう。
FullHD の 1920*1080 なら UI はとても綺麗に表示されます。しかしパフォーマンス面の影響も大きいため、半分の 960*540 程度に抑えておくと負荷も下がります。
この辺りはあなたのゲームのセールスポイントによって決めるのがベスト。
ノベルゲームのようにイラストが重要な要素のゲームであれば Canvas サイズを大きくして綺麗に見えるようにすべきですし、プレイヤーがキャラクターを操作してステージを進んでいくタイプのゲームであれば UI はそこまで綺麗でなくても大丈夫です。

デバイスの画面に応じた伸縮

Unity の Canvas コンポーネントではデバイスの画面に応じて伸縮させることができます。
固定サイズにしておくと画面の小さい端末では UI が見切れてしまう原因となるため、画面サイズに応じてスケールする設定にしておくと良いでしょう。
特に iPhoneX などでは画面の比率が異なるため、セーフエリアに UI が表示されるように無料のアセットなどを使った方が良いかもしれません。

分割の有無

UI の要素ごとに Canvas を分割するかどうかを決めます。
Unity ではその Canvas 内の子要素のうち、どれかひとつが更新されるとその Canvas を再描画するため、パフォーマンス面に影響が出ます。
画面に表示する体力ゲージやスコアなどは頻繁に再描画する必要がありますが、ステージ名などは一度描画すれば再描画する必要がありません。そのような場合には再描画の頻度に応じて Canvas を分割する設計にするといいかもしれません。
とは言うものの最初は無理に分割する必要はなく、まずはひとつの Canvas で UI を表示することをおすすめします。

画像の規格

UI に使用する画像の規格を決めます。ユーザの混乱を避けるためにも、ある程度統一感のある画像にした方が良いです。

サイズ

Canvas サイズを元に画像のサイズを決めていきます。
ボタンや、テキストの背景として使用する画像、アイコン画像など、あらかじめサイズを決めておくことで画面全体で統一感が出ます。

インポート設定

画像をインポートする際の設定についても決めておくと後の作業が楽です。
1 枚の画像から複数のスプライト画像をインポートするのか、画像の圧縮率はどの程度にするのかを決めておきます。画像を圧縮すればパッケージサイズが減りますが、圧縮の品質によっては画面が汚くなるので注意が必要です。
対象の端末がサポートしている OpenGL のバージョンによっては高品質でかつ画像サイズを抑えられる設定を選べるので、あなたがリリースしようとしているプラットフォーム仕様や、端末の OS バージョンと合わせて確認してください。
また、Sprite のインポート設定では 9 スライスを使うと小さい画像でもうまく拡大することができます。

テキストの規格

画面に表示するテキストの規格を決めます。

フォント

文字を表示するフォントを決めます。

フリーフォントでも品質の良いフォントがあるので、プレイヤーにとって見やすく、かつあなたのゲームの雰囲気にあったフォントを選びます。

フォントを選ぶ際にはゲームへの埋め込みが可能かどうか、ライセンスに注意してください。有料フォントの場合は埋め込みに制限されている場合もあります。

文字サイズ

文字のサイズを決めます。

通常の会話文は 24 ポイント、ボタンのテキストは 20 ポイント、説明文のヘッダーは 32 ポイント……といったように文字の大きさを決めておくと設定が楽です。

同じ要素ならなるべく文字の大きさを揃えるとプレイヤーにとっても自然に見えます。

文字色

文字のサイズと同様に色も決めておきましょう。

背景画像の色によっては文字が見えにくい場合もあるため注意が必要です。

男性の約 20 人に 1 人は赤と緑を見分けにくいなど色覚特性が異なることがあるため、そうしたプレイヤーへの配慮も必要です。

詳しくは「カラーユニバーサルデザイン」で検索してもらえるといいかもしれません。

この部分はデザイナーさんに協力してもらっても大丈夫です。

ライティングのクォリティ

綺麗なライティングをするのか、ある程度のライティングにするのか、あるいはライティングはデフォルトのライトだけにするのか、品質を決めます。
2D のゲーム場合はライティングを行わずにスプライトで表現した方が綺麗かもしれません。
3D の場合でも端末のスペックは考慮に入れる必要があります。例えばスマートフォン向けアプリなら、ライティングの品質は抑えてある程度に留めた方が良いと思います。

グローバルイルミネーション

ライティングといえばグローバルイルミネーションです。
グローバルイルミネーション (GI) は光源から直接当たる光だけではなく、物体に反射した光の影響も計算します。
この間接光の影響を計算する必要があるため、パフォーマンス面で影響があるのです。
GI にはゲーム実行中に計算を行うリアルタイム GI と、事前に光の影響を画像で記録しておくベイクド GI があり、それぞれで使用 / 不使用を選択できます。
私の場合、モバイル端末では両方使わないか、あるいはベイクド GI だけ使います。

ライトマップのサイズ

事前に計算された光の影響を画像として保持しているライトマップについて、サイズを決めておきます。
画像としてデータを持つため、サイズが大きければ精細な画像に、サイズが小さければ少しぼやけたような画像になります。
モバイル端末の画面で精細なライトマップが必要なケースはそれほど多くないため、ライトマップは小さくても問題ありません。

プローブの使用

ライトマップとして光の影響を計算しておけるのは、シーン内で Lightmap Static のオブジェクトだけです。
ゲーム実行中、動くオブジェクトに対して間接光の影響を与えるためには Light Probe が必要になります。例えば操作しているキャラクターがある空間を通った時、付近の光の影響を与えたい場合には Light Probe を用意しておきます。
綺麗な画面が売りであるなら GI と合わせて使った方が良いですが、モバイル向けのゲームであれば無理して使わなくて良いと思います。

必要な素材のリストアップ

ゲーム内で使用する素材 (アセット) をリストアップします。
個人でゲームを作っている場合は後からでも追加しやすいですが、デザイナーさんやイラストレーターさん、サウンドクリエイターさんに依頼をする場合は事前にリストアップしておくことが重要です。
そのため、ある程度ゲームの骨組みを先に作ってしまい、素材の数が見積もれるようになってから依頼するのもありかもしれません。
この場合は納期が先になることもあるので、ゲームのリリース日との相談になります。
あるいはアセットストアにある素材を使用するのも便利です。正直に言うと、既に完成しているアセットをすぐに使えるため、ゲームのイメージに合うものがあればアセットストアを有効利用した方が早くゲームが完成します。

グラフィック

画面の表示に関わる素材をリストアップします。

3D モデル

3D のゲームではキャラクターやステージ内のオブジェクトに 3D モデルが必要です。
Unity で作成できる Cube や Sphere を活用することで必要な素材数を減らすこともできます。
自分で 3D モデルを作成する場合には、Maya や Blender などを使います。癖はあるものの、無料で使える Blender が気軽に使えて便利です。

2D スプライト

ドット絵や背景画像などの 2D のゲームで使用するスプライト画像を作成します。
キャラクターの画像はアニメーションさせるために差分の画像も必要かもしれません。
幸い、キャラクターのドット絵やマップチップなどを公開しているクリエイターの方々もいるので、素材を使わせてもらうのも良いと思います。
素材を使う場合は利用規約をよく確認します。

テクスチャ

主に 3D ゲームでオブジェクトの表面に貼るテクスチャ画像を用意します。
草原のテクスチャ、岩のテクスチャ、荒地のテクスチャなど、様々な画像が必要です。
アセットストアで購入するのが一番早いかもしれません。

UI

UI で使用する画像を作成します。
メッセージウィンドウ、ボタンの背景、アイコン画像など、画面イメージに合わせて作成しましょう。
こちらも素材を公開しているクリエイターの方々がいるのでそれを使わせてもらうといいかもしれません。
利用規約を確認の上、それぞれの素材の雰囲気が統一されているかも確認します。別の人が作った素材を混ぜるとキメラのように統一感のない画面になってしまうためです。

オーディオ

オーディオデータの素材をリストアップします。

BGM

ゲーム内で流れる BGM を作成します。
フリー素材を公開しているサウンドクリエイターの方々も多く、検索するとすぐに素敵な素材が見つかります。
こちらも利用規約を確認の上、使用します。
フリー素材は他のデベロッパーも使用していることが多いので、重複を避けたい場合にはアセットストアの素材を使うと良いかもしれません。

効果音

ゲーム内で使う効果音を作成します。
こちらもフリー素材を公開している方々がいるのでありがたく使わせてもらいましょう。
効果音の場合はよほど特徴的でなければ他のゲームで使われていても気にならないと思います。

ゲーム内で扱うデータ

ゲーム内では様々なデータを使用します。
Excel などで元データを作成し、Editor 拡張でインポートするとスマートです。Unity 内でデータを持つクラスを ScriptableObject で作成しておくと実行時に素早く扱えます。
サーバにデータを保存し、実行時に読み込む形であればアプリのアップデートなしでゲームデータを更新できるメリットもあります。
どちらがいいかはあなたのゲームによりますが、最初はアプリ内に持っておくのが楽です。
以下に挙げたのは一例ですが、あなたのゲームではもっと必要なデータがあるかもしれません。
あなたのゲームを一度頭の中で遊んでみるとそうしたデータの種類が分かってくるはずです。

キャラクターのステータス

RPG であればキャラクターのステータスデータを設計する必要があります。
どのようなパラメータが必要なのかは敵キャラと合わせて考えます。
初期ステータスの他に、レベルが上がるごとにステータスがどれだけ上がるか、レベルアップに必要な経験値はいくつか、レベルいくつで魔法を覚えるかなど、設定すべきデータは多いので、あなたのゲームに必要なデータをリストアップしておきます。
その他のゲームでも、プレイヤーの行動を記録するためにデータを保持するクラスを設計する必要があるかもしれません。

アイテムデータ

ゲーム内で使われるアイテムのデータを設計します。
RPG で使用するのがイメージしやすいですが、他のジャンルのゲームでも何らかのアイテムを手に入れてそれを使うことがあるため、必要なアイテムがあればリストアップします。
脱出ゲームであれば脱出のためのアイテム、レースゲームなら加速したり敵を攻撃したりするアイテム、スタミナ制のゲームであればスタミナ回復用アイテムなど、思いついたものはどんどん挙げていきます。

技や魔法のデータ

ゲーム内で使う技や魔法のデータを設計します。
キャラクターのデータや、次の敵キャラのデータとも関連して考えておくと良いでしょう。
ここで具体的な数値を設定するというよりは、技や魔法の種類や威力の目安、コストの目安を考えておくに留めます。

というのも具体的な値はテストプレイでのバランス調整で嫌という程設定できるので、
まずは概要だけ考えておくのが良いと思います。

敵キャラのデータ

ゲーム内で出現する敵キャラのデータを設計します。
敵の種類、強さの目安、ドロップアイテム、到達レベルの目安などを考慮して敵キャラを考えていきます。
弱点を表現するために、付随して属性データなども設計する必要があるかもしれません。
アクションゲームで武器ごとに効く / 効かないを決めるのもいいでしょう。
敵キャラの見た目については、ゲームのテーマや雰囲気に合わせて決めていきます。
敵キャラのグラフィックやモデリングをデザイナーさんにお願いする場合は先に敵の数、種類、見た目を決める必要があります。

お店の品揃えデータ

ゲーム内のお店で買えるアイテムの品揃えを設計します。
店売りするかどうか、アイテムデータと関連させながら考えていくと良いと思います。
ゲームの進行度に応じて品揃えが増えていくようにするとワクワク感が出てきます。

ステージの攻略状況

ステージの攻略状況やストーリーの進捗状況を格納するデータについても設計を行います。
フラグなどがあればそれも含めて考えておきます。

実績データ

プレイヤーがゲームを遊んだ結果を保存しておきたい場合はそれも設計に含めます。
ゲームのプレイ回数、ハイスコアなどによってトロフィーを獲得できる場合はその条件も考えます。

セーブファイルの実装方式

ゲームのデータを保存するセーブファイルをどのように保持するか設計します。
ゲームの進行度は確実に保存しておきたいので、何を保存するか、どのように保存するかはしっかりと考えておきましょう。

ローカルに PlayerPrefs

Unity のデフォルト機能である PlayerPrefs を使って、アプリがインストールされた端末内にゲームデータを保存します。
保存できる型が int、float、string の 3 つなので、簡単なゲームであればこれでも十分です。

ローカルに JSON

Unity では JSON 形式でデータを扱うための JsonUtility というクラスが用意されています。
JSON ではデータを名前と値をペアで保持しており、C# の読み書きの機能 (Stream) と組み合わせることで、セーブデータの保存に使用できます。
保存方法の柔軟性を考えると、慣れてきたらこちらの方法もおすすめです。

ローカルに EasySave

これは AssetStore にあるアセットを使う方法です。
EasySave というセーブに便利なツールがあるので、これを使うと非常に楽です。
PlayerPrefs で保存できる型に加えて、自分で作成した型についても値を保存することができます。
セーブデータの保存先は、アプリの場合はインストールされた端末です。セーブデータ自体を暗号化する機能もあるため、ファイルを書き換えるチートにも強いメリットがあります。

サーバに送信

アプリのバックエンドサーバにデータを送信する方法です。
MMORPG やソーシャルゲームであればこの方法を使います。
プレイヤーの端末内にほとんどデータを持たないため、チートが行われるリスクが減ります。
プレイヤーのデータがサーバ側にあることから、サーバ側の処理でプレイヤーランキングを作成しやすいなどのメリットもあります。
ただし、サーバのセキュリティ、通信のセキュリティ、データの定期的なバックアップなど、考慮すべき点が多いので最初はオススメしません。

広告を表示する場所

ゲーム内で広告を表示する場合は、どこに表示するのか設計します。
ゲームオーバー時に広告を見たら復活、お店で広告を見たらアイテム解禁、広告を見たらヒントを表示など、ユーザにメリットのある方法で広告を表示するのが良いでしょう。
放置ゲームなど、ゲームの区切りが難しい場合はバナー広告などを利用します。

有料アイテムのデータ

ゲーム内課金のアイテムを導入する場合はこれについても設計します。
スタミナ回復のアイテム、操作キャラの追加、ステージの追加など、全てリストアップしておきます。
この時点で価格も決めておきます。
現実世界の 100 円だとスーパーでペットボトル 1 本買えるくらいの感覚ですが、ゲーム内の 100 円は感覚的にかなり重いです。なので課金後にユーザが受け取る価値は大きくなるようにした方が良いですね。
この辺りはあなたのマーケティングプランに関わるので、そちらを軸に考えてみてください。

スプラッシュスクリーン

ゲームを起動した時のスプラッシュスクリーンで何を表示するか決めます。
多くの場合、企業ロゴやチームのロゴを表示しています。
Unity の個人版を使っている場合は「Made with Unity」のロゴが必ず表示されるようになるため、これを外したい場合は Plus 版または Pro 版ライセンスの申請が必要になります。
Plus 版以上であればアセットストアの一部のアセットが 2 割引で買えることもあるため、今後も開発を続けるのであれば申し込んでおいても良いと思います。

ゲームのクォリティを決定

ゲームのパフォーマンスをどのレベルにするかを決めます。
QualitySettings の画面を見ながら決めても良いです。
画像の綺麗さや演出が強化されればされるほどパフォーマンスに影響が出るため、どの部分を残し、どの部分を削るのか、優先度を決めるのも大切です。

ゲームファイルのサイズ

ゲームファイルの目安となるサイズを決めます。プラットフォームごとに使用できる容量が異なるため、制約を確認しておくと良いでしょう。例えばアプリのパッケージは 100MB を越えると Wi-Fi 環境でダウンロードするよう警告が出るため、これより大きくならないようにするのが理想です。
どうしても大きくなる場合は AssetBundle を使って別ファイルとしてリソースをダウンロードさせることを考えます。アプリで特に大きな容量を使うのはテクスチャ、オーディオ、フォントです。これらのインポート設定や圧縮設定についても事前に決めておくとスムーズです。

FPS の設定

ゲーム実行時の FPS(Frames Per Second) を決めます。
リズムゲームや格闘ゲーム、アクションゲームなどタイミングが重要なゲームでは 60FPS を目安とし、その他のゲームでは 30FPS を基本とすると良いかもしれません。
60FPS にすればゲームの見た目がスムーズになる反面、CPU への負荷や電池の消費量が大きくなります。スマートフォンの場合は端末自体が熱を持つことになるため、ゲームプレイに影響が出ることもあります。また、1 フレームあたりの処理時間も短くなるため、スクリプトで工夫しないと処理が間に合わない「処理落ち」が発生します。
30FPS では電池の消費量が抑えられ、処理時間についても多少余裕が出ます。
ゲーム実行中に動的に FPS を変更することもできるので、演出時に 60FPS、通常時は 30FPS(場合によっては 15FPS) に変更して要所要所でスムーズに見せることも可能です。

Target API の決定

アプリの場合はリリース対象の API レベル、Minimum の API レベルを決めます。
Android の場合は Target API レベルと Minimum の API レベルの両方を決め、iOS の場合は Minimum の iOS バージョンを決めます。
各 OS バージョンのシェアについてはデータをまとめているサイトがあるため、そちらを参考に決めると良いでしょう。

アプリストアでは一定の周期ごとに Target API の見直しが必要になるため、ストアからの通知をよく確認するようにしましょう。
また、古い API レベルの端末をサポートするのは工数もかかるので、広い範囲をサポートするのはあまりお勧めできません。Minimum API レベルを下げて多くの端末に対応させたい気持ちは痛いほど分かりますが、多くの API レベルを対象とするとバグも発生しやすいので、なるべく絞った方が良いと思います。

外部サーバの使用

外部サーバを使う場合はその選定と処理内容を決めておきます。
最初のゲームであれば、無理して使う必要は無いと思います。

サーバの選定

どのサーバを使うかを決めます。
既に用意されていたり、資金が潤沢にあるのであれば自社サーバを用意するのが一番です。
多くの場合はクラウドのサービスを使った方が低コストで早いかもしれません。

自社サーバ (オンプレミス)

自社サーバであれば自由にカスタマイズできるので、制約が少ない環境で開発できます。
自社の敷地内にあるサーバですから、物理的なアクセス環境も良いです。
反面、サーバルームの設置や運用コストを考えると値段はかかります。アプリのバックエンドサーバとして使うには少々もったいないですね。絶対に外部に漏らすべきでない情報を扱っているのであれば選択肢に入ります。
小規模アプリ開発者の立場からはオススメしません。

クラウド

AWS や Azure でサーバを借りたり、バックエンドサービスを利用する方法です。
データの配信があるなら AWS や Azure を使い、オンラインランキングやデータの保存が目的ならバックエンドサービスを選択するのが良いでしょう。
ゲームで使う機能がある程度用意されている点から、バックエンドサービスの方が早いかもしれません。
柔軟にカスタマイズしたい場合は AWS や Azure を使います。

サーバ側の処理

サーバを使って何を処理させるのか設計します。
多くの場合はリソースのダウンロードや、ゲームデータの保存になると思います。
サーバとの通信方法や、データのフォーマットについてもあらかじめ決めておきます。
データを保存するなら DB のテーブル設計も必要です。

リソースのダウンロード

ゲーム内で使う画像や音楽をダウンロードさせることができます。
アプリとしてゲームをリリースする場合はアプリのパッケージに含めず、初回起動時やチュートリアル中にダウンロードさせるとスムーズです。
iOS のアプリストアの規約では実行コードのダウンロードが禁止されているので、ダウンロードするファイルに含まれていないか注意が必要です。

ゲームデータの保存

ゲームプレイの結果をサーバに保存します。
バックエンドサービスであれば機能が既に用意されている場合があります。自前で実装するならサーバ側でデータベースを用意してそこに保存する形になると思います。
ゲームからデータを受け取って DB に保存、DB からデータを読み込んでゲームに送信、といった処理が必要です。

イベントでランキング

何らかのイベントを開催してそのランキングを実装したいケースもあります。
バックエンドサービスのランキング機能では更新タイミングが決められていることもあるので、自前で処理した方が柔軟に対応できるかもしれません。

セキュリティ

データの保護について設計します。
ここで考えるのは主にチート対策ですが、この対策は正直に言うとキリがないので、無理に導入する必要がない場合もあります。
他のユーザに影響の出るオンラインゲームやソーシャルゲームなどでは対策が必須ですが、そのソフトウェア内で完結するゲームであればチートした本人のゲーム体験が損なわれるだけなので、目を瞑っても良いかもしれません。
ただし、外部と通信しない場合でも、ゲーム内で課金アイテムがある場合には収益に影響が出るため対策が必要です。

ファイルの暗号化

セーブファイルの暗号化や、敵のパラメータデータなどの暗号化を考えます。
前述の EasySave というアセットを使っていればセーブファイルの暗号化機能が含まれています。
セーブファイルを書き換えることで主人公が無敵になったり、クリアしていないステージをクリアしたことにされたりするため、書き換えが困難になるようセーブファイルの暗号化を行うと良いでしょう。

メモリ上の値の暗号化

メモリ上に展開したデータの書き換えにも対策が必要です。
値き書き換えを検知したらアラートを表示するなどの設計を行います。

通信の暗号化

外部サーバと通信する際にも対策が必要です。
ゲーム内のデータを送信するタイミングで値を書き換えたり、通信を傍受して値を書き換えたりされることもあるため、安全な通信を行いつつ、データを受け取ったサーバ側で値の検証を行うなどの設計が必要です。
対策をしてもそれを破られるイタチごっこにもなりがちなので、大規模なゲームであればセキュリティの専門企業に対策を依頼することも選択肢に入ります。

コーディング規約

ゲーム内の処理をプログラミングしていくにあたって、コーディング規約を決めます。
チームで開発している場合はもちろんのこと、自分一人で開発している場合にもぶれることが無くなる為、決めておくことをオススメします。
チーム内での制約が特にないのであれば、Microsoft 社の C# コーディング規約を使うのが良いと思います。

プログラミング部分の内部設計

上で決めたゲームの振る舞いをプログラムの形で実装するために、どのように作っていくか設計を詰めていきます。
理想を言えばこの設計フェーズでバッチリ決まってしまえばいいのですが、動かしながら確認することも多々あるのでアジャイル方式で進めるイメージが良いと思います。
UML で記述するようにはしていますが、個人で開発しているのであれば全て完璧に設計するというよりは、だいたい決まったら開発に入ってしまった方が早いです。
下に挙げているものも全て作る必要はなく、あなたのゲームの開発に必要な部分だけ作るといいかもしれません。
UML に慣れていない場合は開発中の自分が見返して分かる形で書いておけば大丈夫です。
チームで開発している場合はある程度固めておくと、進捗も分かりやすくなります。

処理の流れ

フローチャートの形でゲームの流れを決めます。
ゲーム全体の流れと、各シーンでの流れ、それぞれを図にしておくと分かりやすいです。
処理の開始、分岐ポイント、ループ、処理の終了が分かるように描かれていれば大丈夫です。

クラス図

ゲーム内で使用するクラスを設計します。
シーン内のオブジェクトにアタッチするクラスや、ゲーム全体で使用するクラス、データを格納するクラスなど、役割と使用場所に応じてクラスを設計していきます。
クラス同士の関連についてもここで設計できると開発がぐっと楽になります。
上で決めた振る舞いが実現できるように考えていきます。
チーム内で分業する場合は特に気合いを入れて設計すべきポイントです。

ユースケース図

ユーザの操作に対するアクションを図で表します。
ユーザがボタンを押した時にどのような振る舞いをするか、キャラクターを動かした時にどう動くかなど、ユーザの視点でゲームの振る舞いを明確化します。

シーケンス図

ゲーム内のクラスやオブジェクトがそれぞれどのように相互作用しているのか時間を軸に図に表します。
何らかの処理が終わったら別のオブジェクトに通知することも多々あるため、これを図にしておくと分かりやすいです。

ゲームオブジェクトにアタッチするコンポーネント

シーン内のオブジェクトに関しては、どのオブジェクトにどんなコンポーネントをアタッチするかも考えておくと後の作業が楽になります。
個別のオブジェクトに関して設計を行うというよりは、キャラクターには Rigidbody と Collider をアタッチする、動かないオブジェクトでは Collider だけアタッチする、といったように種類ごとにどんなコンポーネントが必要なのかを決めておくイメージです。

サーバと通信するタイミング

外部のサーバと通信を行う場合は、そのタイミングも設計しておきます。
通信を行う前にデータをまとめて、送信の準備をして……と処理を入れる必要があるため、これも明確にします。
シーケンス図に含められるのであれば含めてしまった方が分かりやすいかもしれません。
サーバからの応答を待ち合わせる処理も忘れずに考えておきます。

ホームボタンの動作

スマートフォンにはホームボタンや戻るボタンが搭載された端末があります。
これらのボタンが押された時の動作についても検討します。
iPhone についてはホームボタンが押されたらゲームを中断したりポーズ画面にするなどの処理を入れると良いと思います。
Android については戻るボタンがあるため、ダイアログでキャンセルボタンと対応させるなどの処理があると良いでしょう。
この辺りはゲームに合わせて最適な処理を考えます。

準備

開発に入る前に準備を行います。
既に終わっているものもあるかもしれませんが、チェックリストとして以下を確認すると良いかもしれません。

Unity ID の登録

Unity ID の登録を行います。
Unity の Web サイトからすぐに作れるのでまだ作成していない場合は作っておきます。

Unity Hub のインストール

Unity のエディタを使うにあたって、Unity Hub をダウンロードしておくと便利です。
Unity エディタのインストールも Unity Hub 内で行えるので、先にこちらをインストールします。

Unity エディタのインストール

Unity Hub を使って Unity エディタをインストールします。
特にこだわりがなければ最新バージョンか、その一つ前のバージョンが良いと思います。

サポートモジュールのインストール

Unity のインストール時に iOS ビルド、Android ビルドを作成するためのサポートモジュールをインストールします。
忘れても Unity Hub から後でインストールすることもできます。

ライセンスの発行

スプラッシュスクリーンで「Made with Unity」の表示を外したい場合は Plus 版か Pro 版のライセンスが必要になります。
Unity の Web サイトからライセンスを購入し、Unity Hub を使って適用します。

Unity Ads への登録

広告を表示する場合は Unity Ads の登録を行います。
Unity Ads のページからアカウントを作成します。

(iOS) Apple Developer Program への登録

iOS 向けにリリースする場合は Apple Developer Program に登録する必要があります。
年会費がかかるのでその点には注意が必要です。

(Android) デベロッパーアカウントの登録

Android 向けにリリースする場合は Google Play デベロッパーアカウントに登録する必要があります。
こちらは最初の登録時に料金がかかるだけです。

販売者アカウントのセットアップ

有料アプリとしてリリースする場合や、アプリ内課金を導入する場合は販売者アカウントをセットアップする必要があります。
Android の場合はお支払いプロファイルを作成します。
iOS の場合は口座情報などの登録を行います。
販売者の情報 (販売者の名前や住所) が公開される点に留意します。

外部サーバの準備

外部サーバを使用する場合は、設計フェーズで決めたサーバの準備を行います。

ドメインの取得

アプリ情報を掲載したり、プライバシーポリシーを掲載する Web サイト (ブログなど) を利用するためのドメインを取得します。
ドメインと合わせて Web サイト用のサーバもレンタルしておくと作業がスムーズです。
ブログにする場合は既存のテンプレートを使うと素早く開設できます。

(Mac) Xcode のインストール

iOS でアプリをリリースするためには Mac 専用の Xcode という IDE が必要です。
こちらも Mac App Store からダウンロードしてインストールしておきます。
ゲームの開発自体は Unity で行いますが、Unity が出力したプロジェクトからアプリのパッケージを作成する時に使います。

Android Studio のインストール

Android Studio をインストールしておきます。
Unity では apk ファイルを直接ビルドできますが、シミュレータを使うにあたって Android Studio を準備しておくと便利です。

開発

準備が終わったら実際に開発を行なっていきます。
このパートはゲームを作る上での醍醐味ですね。

設計に沿った開発

設計フェーズで決めたことに沿って開発していくとスムーズです。

設計に戻ってもいい

開発フェーズに入って、設計を変更すべき点が見つかったら設計に戻っても大丈夫です。
「作ってみたらこっちの方が良かった」といったことはよくあります。
大切なのは「ユーザにとって楽しめるゲームになっているか」なので、この目的に合わせて設計を修正するのは良いと思います。
大きく変わりそうであれば、リリース後のアップデート項目に回す決断も必要です。

日誌をつける

開発中はどのような作業をしたのかを日誌につけておくことをお勧めします。
設計変更が入った時にその意図を残せるので、他の作業への影響も分かって便利です。
プロジェクトが終わった後に振り返りを行うためにも情報を残しておくと今後の開発に繋がります。
私の場合はノートにその日の作業予定、気付いたこと、作業実績などを残しています。(開発に熱中するあまり書き忘れる日もありますが……)
理想は開発を行なった日毎に書ければ良いですが、「毎日書かなきゃ!」と追い詰められるよりは、気付いた時に情報を残すくらいの感覚で気楽に書くと良いでしょう。

Project Settings の設定

設計で決めたクォリティに合わせて Quality Settings を設定します。
Player Settings についてもターゲット API レベルなどを設定していきます。
リリース前に Bundle Identifier やバージョン、アイコンの設定など、再度確認を行います。

Quality Settings

Very Low から Ultra の中から適切なクォリティ設定を選択します。
個別の設定は必要に応じて変更しますが、多くの場合プリセットの中から選べば大丈夫だと思います。

Player Settings

アイコンの設定、画面の向きの設定、Bundle Identifier やバージョン、ゲームデータの保存場所など、設定すべき項目がたくさんあります。
iOS 特有の設定と Android 特有の設定もあるので忘れずに設定しておきます。

各シーンの作成

設計で決めたシーンを作成していきます。
パーツを共有できるシーンがあれば、シーンを複製したり Prefab を活用することで素早く構築できます。

オブジェクトの配置

シーン内にオブジェクトを配置していきます。
フィールドだったり、部屋だったりとゲームに合わせて舞台を作成します。
オブジェクトの位置は Transform を確認しながら数字で決めていくとうまく揃います。
自然さを演出したい時には高さだけ揃えて平面上の位置をばらけさせるなどの工夫をしてみましょう。
3D モデリングをデザイナーさんにお願いしていてまだ納品されていない場合は、オブジェクトを配置して仮置きしておくと差し替えがスムーズになるかもしれません。

コンポーネント設定

シーンに配置したオブジェクトにコンポーネントやスクリプトをアタッチします。
メッシュのあるコンポーネントに Collider をアタッチしたり、キャラクターオブジェクトに Rigidbody をアタッチしたりとあなたのゲームに合わせて設定します。
シーン全体の動作を管理する Manager 用スクリプトがある場合は Create Empty を使って空のオブジェクトにアタッチすると分かりやすいです。
クラス設計に応じて、同じゲームオブジェクトにアタッチした方が良い場合はまとめてアタッチし、分けた方が良ければ別のオブジェクトを作成してアタッチします。

ライティング設定

オブジェクトの配置が終わったら、光源を配置して画面の明るさを調整します。
この調整は Game タブでカメラの画像を確認しながら行うと良いでしょう。
動かさないオブジェクトについては Static にしてライトマップをベイクさせます。
設計時に決めたクォリティに沿ってライトマップのサイズ、間接光のサンプル数を設定します。プローブについてもシーン内に配置しましょう。

BGM、SE 設定

シーン内で再生する BGM や SE を準備します。
コンポーネントにアタッチしてシーン起動時に自動的に再生する方法でも良いですし、制御用のスクリプトと一緒にアタッチしてそちらから再生する方法も良いと思います。
個人的には制御用のスクリプトから再生するのが好きです。
というのも、コンポーネントから自動再生させた場合、シーン遷移時に「プチっ」と音が急に途切れてしまい気になるためです。
SE の方はシーンの起動時に鳴らすわけにはいかないので、スクリプトから再生タイミングを制御します。

UI 配置

シーン内で情報を表示するための UI を配置します。
設計時に決めた Canvas の設定を行い、必要な UI を配置していきます。
こちらもデザイナーさんにお願いした画像がまだ納品されていなければ、デフォルトのスプライト画像などで仮置きをしておきます。

ボタンの動作設定

UI と関連してボタンも配置します。
ボタンに関しては押された時に呼び出すメソッドの設定も忘れないようにします。
ボタンが押された後に連打されないような制御も必要になるのでこちらも忘れずに。

キャラクターのアニメーション設定

2D のゲームであれば、Unity のシーン内でアニメーションの設定を行います。
スプライト画像の切り替えによって歩行のアニメーションを表現したり、UI のテキストウィンドウでメッセージ送りのアイコンを動かしたりと幅広く表現できます。
3D モデリングの場合は Unity の中でアニメーションを付けるのは難しいため、モデリングソフトの方でアニメーションを作成してインポートするのが良いでしょう。

スクリプト作成

設計にしたがってスクリプトを作成していきます。
シーン作成と並行して行うことも多いかもしれません。
テストフェーズと分けてはいるものの、実際にはある程度スクリプトを更新するたびに Play ボタンを押して動作確認しています。
また、コメント文には処理の意図を残して、テスト時やアプリのアップデート時に分かるようにします。

シーン内の制御スクリプト

シーン内の管理や制御を行うスクリプトを作成します。
個々のオブジェクトに対する制御だったり、シーン管理の Manager 系スクリプトだったりとシーンの中で使うスクリプトは多くなります。
もし Manager 系のスクリプトが肥大化するようであれば、リファクタリングやクラス設計の再検討を考えます。

グローバルなスクリプト

ゲーム全体で使用する Static なクラスもいくつか作成する必要があるかもしれません。
量が多くなるとメモリを圧迫し始めるため、必要な時にインスタンス化して使うのも大切です。

セキュリティ用スクリプト

セキュリティ面での対策を行うスクリプトを作成します。
メモリ上で値を持つ時に暗号化したり、セーブファイルを暗号化する処理を入れます。
暗号化のためのキーとなる文字列もアプリ内に含めないといけないので、完全な防御は難しい点に留意します。
設計でセキュリティ対策を行わないようにしているのであればここは不要です。

エラー発生時の処理を実装

スクリプト内でエラーの発生しうる処理については例外処理を入れます。
ユーザのデータ保護はとても大切なので、セーブデータが壊れないように気を配ります。
セキュリティ面での対策を行っているのであれば、不正な値や操作を検知した時の処理も忘れずに実装します。

サーバ側の処理を実装

具体的な処理はゲームによりますが、ゲームからのデータ送信、受け取ったデータの DB への保存、リクエストに応じたゲームへのデータ送信などの処理が必要になるので、設計に応じて実装します。
設計ができていれば分業しやすい部分でもあります。

ゲーム内で扱うデータの作成

設計で決めたデータの種類に応じて、それぞれ具体的な値を決めていきます。
設計部分でも触れましたが、敵の強さやアイテムの値段など、値のバランス調整はテストプレイを通して決めていくことになります。
まずは調整のベースとなる値を設定していきましょう。

外部データのインポート用ツールの作成

Excel でデータを入力している場合は、Unity 内に取り込むツールも作成します。
Editor 拡張スクリプトなどを作成する必要があるものの、データをコピペして Unity で設定を行うよりは遥かに楽です。

JSON などでデータを書き出す場合はそのファイルを Unity にインポートして実行時に値を取り出す形式でも良いと思います。
この場合も値を取り出すための処理はスクリプト内に含める必要があります。

素材の作成

ゲームで使用する素材を作成していきます。

グラフィック

グラフィックに関する素材を作成します。
自作はどうしても時間がかかるので、フリー素材を使うかアセットストアの素材を使うとゲームが早く完成します。
オリジナル素材を使ったゲームはある程度慣れてきたらにする、と決断することも重要です。

3D モデル

3D モデルのメッシュやアニメーションを作成します。
素材自体の作成に加えて、Unity にインポートした後に Prefab 化したり、アニメーションの切り替え制御を行ったりの作業も必要になります。

2D スプライト

2D のスプライト画像を作成します。
自分で作成する場合はドット絵ツールなどで作成していきます。
幸い、素材を作成されている方が多いので、公開されている素材を使ってスピーディに開発できます。
検索する場合は「ドット絵　素材」「ドット絵　ツクール」「ドット絵　ウディタ」などのワードがおすすめです。

テクスチャ

テクスチャ画像を作成します。
メッシュに貼り付ける場合は UV などの知識も必要になるので、3D モデリングができる方のサポートを得られると良いかもしれません。
手っ取り早く作る場合は、アスファルトや砂利、海の写真をスマホで撮ってきて、Photoshop などを使ってループ素材にします。
Photoshop があればノーマル画像も作りやすいので便利です。
こちらもテクスチャを公開されている方がいるので探してみると良いでしょう。

UI

UI で使用する画像を設計時に決めた規格に沿って作成していきます。
こちらも RPG ツクールやウディタ向けに素材を作っている方が多いので素材を見つけやすいと思います。

オーディオ

オーディオデータを作成します。
作曲などはある程度慣れていないと時間がかかるので、ゲームの完成を優先させるなら人にお願いするか素材を使います。

BGM

ゲームの雰囲気にあった BGM を作ります。
素材を使う場合はループ再生に対応しているかを確認します。
無料で使える『Audacity』などの波形編集ソフトを使えば簡単にファイル形式を変換できます。ファイル形式の変換だけであれば iTunes でも可能です。
Unity にインポートする際のファイル形式は mp3 か ogg が良いと思います。
wav 形式は容量が大きすぎてアプリのパッケージサイズに影響があるため、なるべく避けた方が良いです。Unity では音質を下げて容量を減らすこともできるので、そうした場合は耳で聞いて音質に問題ないかよく確認します。

効果音

ボタンを押しした時や、プレイヤーが敵を倒した時など、場面に合わせて再生する効果音 (SE) を作成します。
素材を使う場合、BGM と同様に wav ではなく mp3 か ogg に変換すると良いでしょう。
こちらも無料で使える『Audacity』などのソフトを使えば簡単にファイル形式を変換できます。

シーン遷移の処理を設定

シーンの切り替えはスクリプトから行います。
シーン切り替え用のスクリプトを作っておけば、各シーンから呼び出すだけで切り替えられるので便利です。
シーンの名前は string 形式で渡すため、static なクラスなどで一括でシーン名を管理しておくと変更に強くなります。

ビルドへの登録

スクリプトからシーンを切り替えるためには、遷移先のシーンを Build Settings の『Scenes In Build』に登録しておく必要があります。
ここに登録していないシーンをスクリプトから読み込もうとするとエラーが出るので、忘れずに設定しておきます。
また、ゲーム起動時に最初に読み込まれるシーンは『Scenes In Build』の一番上にあるシーンなので、順番も気を付けます。

ビルド

ある程度動作確認が済んだらビルドします。
スクリプトのコンパイルをするだけでは見つからなかったエラーが出ることもあるので、早めにビルドを試しておくと対応が楽かもしれません。
また、ビルドレポートを確認して出力されたプロジェクトやパッケージのサイズを確認します。どのリソースがどれだけの容量になっているかを確認できるので、過度に大きなファイルがあれば容量削減を行います。
特に大きくなりがちなのはテクスチャ、オーディオ、フォントです。

プライバシーポリシーの作成

アプリをリリースする際、広告を使用している場合はプライバシーポリシーを掲載したページの URL が必要になります。
広告 ID やその他ユーザのプライバシーに関わる情報の取り扱いを明らかにする必要があるので忘れずに作成しておきます。
広報用のブログや既に開設してあるブログがあればそこで掲載すると良いでしょう。なければいっそ作ってしまっても良いかもしれません。

テスト

開発が終わったらテストに入ります。
実際には開発を行いながら動作確認や単体テストまでやってしまうことの方が多いかもしれません。
ゲーム開発の面から言うとテストプレイがメインになります。

テストの流れ

一般的な IT でのシステム開発では以下の流れになっています。
完全にこれに従うよりは、うまく使えそうな部分を抜き出してみると良いと思います。

単体テスト

動作確認や分岐のチェックなどがここに該当します。
正常動作だけではなく、エラー発生時の動作についても意図した通りになっているか確認しましょう。
分岐の条件については同値分割、境界値分析を行います。
理想的にはテストコードを書いて自動化したいところですが、当然ながらコストは大きくなるため、手動で何度も行うには辛いテストに絞ってテストコードを作成します。
リリースした後、ユーザの反応を待っている間にテストコードを仕上げるのもありです。

結合テスト

シーン全体の動きを確認するのがここに該当します。
コンポーネント間のメッセージのやり取りや、メソッドの呼び出しタイミング、値の受け渡しがうまくいっているか確認します。
シーンの遷移なども正しく動いているか確認します。

システムテスト

ゲーム全体の動作確認であるテストプレイがここに該当します。
全体が設計通りに動いているか、企画で考えていた面白さが表現できているかなどを確認します。
後述の『テストプレイ』の項目で深掘りします。

パフォーマンステスト

ゲームを実行した時のパフォーマンスを確認します。
期待通りの FPS になっているか、メモリ使用量は適切か、といったパフォーマンス面をチェックします。
確認には Unity の Profiler や、Xcode の Profiler を使います。
FPS に関しては、1 秒間に呼ばれたフレーム数をカウントし、画面に表示するスクリプトを作っておくと便利です。インターネットで検索するとサンプルコードを公開している方もいるので、参考にしてあなたのゲーム用に作ってみると良いでしょう。

テストプレイ

Unity のエディタ上で、あるいは実機にアプリをインストールしてゲーム全体を通してテストプレイを行います。
非常に重要な部分なので、可能であれば友人や仲間に協力してもらうことも検討します。

ゲームのバランス調整

敵の強さ、ステージの難しさ、アイテムの入手確率など、プレイを通して確認します。
開発者自身がプレイしていて難しく感じる部分は、ユーザにとってはその数倍難しく感じる部分です。
自分が思うよりもう少し簡単にした方がいいかもしれません。
ここで気付いたことは細かくメモしておき、あとで俯瞰的に調整を行います。

ゲームプレイの快適さ

ロード時間の長さや演出の長さ、レスポンスなどでストレスを感じる部分がないか確認します。
パフォーマンスが許す限り、なるべくテンポ良くゲームが進んでいくとプレイしていて気持ち良いです。
キャラクターを操作する場合も、画面に表示するコントローラ用の UI が適切な大きさか、レスポンスが良いかなどをチェックしましょう。
この辺りも自分以外の人に触れてもらうといいかもしれません。

面白さ

企画で意図した面白さが表現できているかを確認します。
プレイヤーが熱中するであろうポイントで自分や友人が熱中できているかどうか、人に楽しさを伝えたくなるかの観点でプレイします。
最低限自分で面白いと感じるレベルにはしたいところです。そうすれば自分と同じようなユーザは熱中してくれるためです。
不思議なもので、「多くの人に楽しんでもらう」という考え方よりも「こういうターゲットに熱中してもらいたい」と絞ったゲームの方が心に刺さりやすいんです。
リリース後は実際にユーザに遊んでもらうことで反応がもらえるため、その意見も取り入れていくとより良いゲームになっていきます。

想定外の動き

RPG であればキャラクターに延々と話しかけるとか、レースゲームであれば壁に向かってぶつかってみるとか、通常プレイでやらないような動きを確認します。
RPG だとフラグによる管理ができていないことを検知できたり、レースなら壁に Collider がなくて異次元に落ちていくことを検知できたりと、思わぬ穴があったりするのでこの確認は重要です。
作者本人がプレイしていると無意識の内に正しい動きをしてしまいがちなので、できれば他の人にお願いしたいところです。

リグレッションテスト

テストを通して何か修正を行なった場合はリグレッション (回帰) テストを行います。
バグ修正などに伴って、もともと正常に動いていた動作を変えてしまっていないかを確認するのも重要です。
バグ修正はそれ自体が新しいバグを産みやすいので、既存の動作に影響を与えていないことを確認するため、一度テストした項目を再度確認します。
このリグレッションテストでは自動的にテストを行なってくれるテストコードが効果を発揮します。

テストコードの作成について

上でも少し触れましたが、自動的にテストを行なってくれるテストコードを書いておくとテストが楽になります。
Unity では Test Runner の機能があるため、テスト用のスクリプトを書けば自動的にテストすることが可能です。
特に分岐テストや境界値分析では手動でやるのも負担が大きいので、自動化できると嬉しいポイントです。
しかしながら、テストコードを書く場合でもテストコード自体のテストが必要になったりと工数がかかるため、全てのテストを自動化するのは現実的に厳しいものがあります。
そのため、手作業で行うには負担が大きいテストに絞ってテストコードを書くなど、優先順位を付けて作業をすると良いでしょう。
初回リリースまでの間は手動で頑張って、リリース後にユーザの反応を待ちつつ、既存のテストケースを自動化していくのもありです。
特にアプリのアップデート時には既存の動きに影響が出ると大変なので、そうした部分で品質を保つためにも導入を検討します。

シミュレータによるテスト

Unity のエディタでのテストが終わった後は、各 OS のシミュレータを使って起動確認や画面サイズの確認を行います。
リリース対象の全 OS バージョンで実機を用意できるのが理想ですが、それは難しいのでシミュレータで起動確認をしておきましょう。
iOS のシミュレータは特に動作が重いため、アプリが起動すること、UI がセーフエリアに表示されることを確認できれば OK くらいの感覚で良いと思います。

実機テスト

実機向けにビルドし、端末にインストールしてテストを行います。
自分の持っている端末では最初から最後まで通して遊んでみることをおすすめします。
操作性や端末が熱くならないかなど、Unity エディタでのテストでは確認できない部分を注意深く確認します。

リリース

テストが終わったらいよいよリリース準備を行います。ここでは主にアプリのリリースを扱います。

(Unity Ads) 広告のテストモード設定を外す

広告を導入している場合は、テストモードの設定を外します。
テストモードがオンのままリリースしてしまうと、製品版にテスト広告が表示され、収益が上がらず悲しい思いをする羽目になります。
Unity Ads の管理画面ではテスト用端末の登録ができるので、リリース前に自分の端末をテスト端末として登録し、アプリに導入した広告のテストモードを外しましょう。

ProjectSettings の Player 設定を確認

リリース前に ProjectSettings から Player 設定を再確認します。

アイコン

iOS 向け、Android 向けにアイコンを準備します。
各ストアで決められているサイズを確認し、それぞれ設定しましょう。
Android 向けにアイコンを作る場合は Android Studio に入っている『Image Asset Studio』が便利です。

Product Name

アプリの Product Name を確認します。
デフォルトだとプロジェクト名が入っています。ローカライズファイルを用意していない場合はこの名前がアイコンの下に表示されます。

Company Name

Company Name を確認します。
チーム名があればそれを使い、なければ自分の名前を入れます。

スプラッシュスクリーン

スプラッシュスクリーンの設定を確認します。

画面の向き

縦でゲームを遊ぶのか、横でゲームを遊ぶのか確認します。
意図しない画面の回転でレイアウトが崩れることもあるため、縦か横に固定すると安心です。

Bundle Identifier

アプリのパッケージを一意に識別する名前が入力されているか確認します。
デフォルトでは「.com」のドメインになっているので、あなたが取得したドメインに合わせて変更されていることを確かめます。

バージョン、ビルド番号

バージョン名やビルド番号を確認します。
初回リリースではバージョン 1.0.0、ビルド番号 1 のようにしておけば問題ありません。
アップデート時にはビルド番号を前回のものより上の数字にする必要があります。

サポート OS バージョン

どの OS バージョンをサポートするのか確認します。
設計で決めたバージョン範囲がサポート対象となっていなければ設定を行います。

(Android) パッケージへの署名

キーストアを使っている場合にはその秘密鍵で、App Signing を使っている場合にはアップロード鍵で署名していることを確認します。

リリース用ビルド

Build Settings の中で、各プラットフォームのリリースビルドを行うようになっていることを確認します。
Android の場合は apk パッケージを出力し、iOS の場合はプロジェクトフォルダを出力します。

パッケージのアップロード

Android では Developer Console で『アプリの作成』からアプリの情報を作成し、apk ファイルのアップロードを行います。
iOS では出力されたプロジェクトを Xcode で開き、Xcode の中から検証とアップロードを行います。
その前に Apple の Developer 向けサイトでログインし、配布用の証明書を作成していることを確認します。
App Store Connect では『新規 App』からアプリの情報を作成します。
これらが済んでから Xcode でアップロードを行うとスムーズです。

ストアに掲載する情報の準備

それぞれのストアに作成したアプリの情報にアイコンや画像などをアップロードしていきます。

ストアアイコン

ストアの指定にしたがってストアアイコンを作成します。
iOS の場合はアップロードしたパッケージに含まれているストア用アイコンが使われます。

プレビュー画像

アプリの内容が分かる画像を数枚用意します。
文字を入れても良いので、ユーザが目を止めてくれるようなキーとなる画像をアップロードすると良いでしょう。

プレビュー動画

可能ならアプリの内容が分かる動画も用意します。
あった方が良いのは間違いありませんが、それでリリースが遅れそうであれば無理に作らなくても良いかもしれません。

ストア掲載文

アプリの内容を説明した文章をストア掲載文として入力します。
単純な機能の羅列をするよりも、その機能があることでユーザがどう楽しめるのかを中心に書くと良いです。

コンテンツのレーティング

アプリの対象年齢を決めるために、レーティングのアンケートに答えます。
それぞれの答え方についてはストアのヘルプにあるので、それを参考にすれば大丈夫です。

価格・配布先の国を設定

企画時に決めた価格、配布先の国を設定します。

プライバシーポリシーの URL を設定

作成したプライバシーポリシーのページの URL を設定します。
Android の場合は広告を入れている場合必須項目です。

(iOS) 輸出コンプライアンスへの同意

iOS では輸出コンプライアンスへの同意が必要になります。
この点についてはあなたのゲームに応じて答え方が変わるため、情報をよく確認してください。

(iOS) 審査への提出

情報が揃ったら審査に提出します。
早ければ 2 日程度、長ければ 1 週間程度かかります。
審査に提出する前に、ガイドラインに沿って開発できているか、再確認すると良いです。
審査に提出する際には、審査に通った後すぐに公開するか、日程を指定して公開するかを選択できます。この点はあなたのプランにしたがって選択します。

(Android) アプリの公開

Android の方では審査なしでアプリを公開することができます。
リリース日をあらかじめ決めてあるのであれば、iOS 版と同時にリリースすると良いでしょう。公開には数時間程度かかります。

アップデート

リリース後はユーザの反応を見ながらアップデートを行います。

広報

リリースの次は広報です。
可能ならリリース準備中から事前に周知できると良いです。
iOS のアプリなら審査に提出している間にリソースを準備できるとスムーズですね。

広告用リソースの準備

広告として使う動画や画像を準備します。

動画

動画広告を準備しておくとゲームの内容が短い時間で分かりやすくなります。
スマホで遊んでいる様子を別のスマホのカメラで撮る、なんて技をやっている方もいたので、おしゃれな動画を作るよりも遊んでいる様子が分かる動画の方が良いようです。

画像

出稿する広告のサイズに合わせて画像を作成します。

広告用テキスト

短い文章でゲームの楽しさを伝えます。この点はなかなか難しいのですが、ゲームのキーワードとなる要素を入れます。

広告の掲載

アプリのダウンロード数を増やすなら広告を出すのが一番早いです。
いくら使って、どれくらいダウンロードされた、というのが数字で出るため改善がしやすいんです。
最初に少額 (1 万円〜 5 万円程度) 使ってユーザに遊んでもらい、その反応を見てアプリやストアの掲載文を変えていくと良いでしょう。ある程度反応をもらって、レビューも好評になってきたら広告費を一気に投入するのがベストな流れです。

Google 広告

Android アプリや Developer Console と相性が良いです。
予算を決めて、リソースをアップロードしておけば AI が自動的に入札してくれるので、結構簡単に出稿できます。
アメリカのストアでアプリを公開していないときに広告を出そうとしたら「アプリが見つからない」と言われたことがあったので、日本語であってもアメリカのストアで公開しておいたほうが良いかもしれません。

Unity Ads

自分のゲームの中に Unity Ads の広告を表示するだけではなく、自分の広告を他の人のゲームで表示してもらうことができます。
値段も多少安価で出せるのでおすすめです。

Facebook 広告

ユーザの住んでいる地域や年齢、性別などで絞って広告を出すことができます。
ゲームのターゲットにリーチしやすいのがアドバンテージです。

Twitter 広告

フォローしていない人にも宣伝できるのが大きいです。
広告ツイート自体もいいねやリツイートができるので、うまくいけば一気に広がります。

その他のアドネットワーク

代表的な広告媒体を紹介しましたが、他にも広告を掲載できるメディアはたくさんあります。
いくつかのアドネットワークを使ってテストしながら、効果の高いメディアに投資すると良いと思います。

Twitter で宣伝

ある程度繋がっている人がいるのであれば Twitter で宣伝するのも良いです。
開発者の人たちと繋がっていると、遊んでくれて感想を送ってくれたりもするので非常にモチベーションが上がります。
この点はお互い様なので、彼らの作品もいいねしたりリツイートするとお互いにモチベーションが上がり、いい作品を作れるようになると思います。
また、Twitter で開発者の人たちと繋がるもうひとつのメリットとして、ゲーム制作の勉強になる点があります。
テクニックなどを画像付きでツイートしている人もいるので、多くのことを学べます。
もちろん、彼らの作ったゲームをプレイすることでも学べることが多々あります。

ブログの作成

それほどお金がかからず、かつ作ったゲームをポートフォリオとして掲載できる点でブログは優れています。
ブログだけでゲームのダウンロード数を増やすのは至難の業ですが、ゲーム作成の実績からお仕事が入ってくるかもしれません。
そのお金を広告費として使えば、人に貢献しつつ自分のゲームを宣伝できるという素敵なスパイラルになります。
この攻略チャートを見てお気付きのように、ゲームをリリースするまでには多くの作業があるので、一度ゲームをリリースしたことがあると、開発を行うときに全体を見通して開発できるようになります。
この点はお仕事を受ける時に有利に働きます。

終わりに

『一瞬で全体像を把握する　ゲーム公開までの攻略チャート』は以上となります。
もしかしたら、この攻略チャートを読みながら、あなたの頭の中では作りたいゲームのイメージが出来上がっていったかもしれません。
もしそうなら今すぐ頭の中のイメージを紙に書き留めておきましょう。
あなたの心にしたがってペンを走らせれば、企画部分が出来上がってしまうことでしょう。
そうなったら設計に入りたくなるかもしれません。
フォーマットにとらわれず、ノートにどんどんゲームの設計を書いてしまいます。
ある程度設計が決まったら、あなたの手は自然と Unity を開いていて、頭の中にあったゲームがどんどん形を持つようになって……。
「ゲーム公開」というエンディングを見られるかどうかは、あなたのこの後の努力にかかっています。

商標について

注意事項

本書の内容についてご意見、ご質問等がありましたら todo のブログ『エクスプラボ』からお問い合わせください。
エクスプラボ
https://ekulabo.com/